

conference

proceedings

2nd Conference on Domain-Specific Languages

*Austin, Texas, USA
October 3–5, 1999*

**Sponsored by
The USENIX Association**

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

**In Cooperation with
ACM SIGPLAN and
ACM SIGSOFT**

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
WWW URL: <http://www.usenix.org>

The price is \$20 for members and \$24 for nonmembers.
Outside the U.S.A. and Canada, please add
\$11 per copy for postage (via air printed matter).

Past DSL Proceedings

Conference on Domain-Specific Languages	1997	Santa Barbara, California, USA	\$20/24
---	------	--------------------------------	---------

© 1999 by The USENIX Association
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN 1-880446-27-8

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

**Proceedings of the
2nd Conference on
Domain-Specific Languages
(DSL '99)**

**October 3–5, 1999
Austin, Texas, USA**

Conference Organizers

Conference Chair

Thomas Ball, *Bell Laboratories, Lucent Technologies*

Program Committee

Tim Bray, *Textuality*

Charles Consel, *Irisa/University of Rennes*

Mary Fernández, *AT&T Labs-Research*

Paul Hudak, *Yale University*

James R. Larus, *Microsoft Research*

Doug Lea, *State University of New York at Oswego*

Jay Lepreau, *University of Utah*

Brad A. Myers, *Human-Computer Interaction Institute,
Carnegie Mellon University*

Todd Proebsting, *Microsoft Research*

David S. Rosenblum, *University of California, Irvine*

Michael Schwartzbach, *University of Aarhus*

Invited Talks Coordinator

Carlos Puchol, *Bell Laboratories, Lucent Technologies*

The USENIX Association Staff

Contents

2nd Conference on Domain-Specific Languages

October 3–5, 1999
Austin, Texas, USA

Message from the Chairv

Index of Authorsvii

Sunday, October 3

Testing and Experience Reports

Session Chair: James R. Larus, Microsoft Research

Using Production Grammars in Software Testing1
Emin Gün Sirer and Brian N. Bershad, University of Washington, Seattle

Jargons for Domain Engineering15
Lloyd H. Nakatani, Mark A. Ardis, Robert G. Olsen, and Paul M. Pontrelli, Lucent Technologies USA

Slicing Spreadsheets: An Integrated Methodology for Spreadsheet Testing and Debugging25
James Reichwein, Gregg Rothmel, and Margaret Burnett, Oregon State University

Optimization and Extensibility

Session Chair: Mary Fernández, AT&T Labs—Research

An Annotation Language for Optimizing Software Libraries39
Samuel Z. Guyer and Calvin Lin, University of Texas at Austin

A Case for Source-Level Transformations in MATLAB53
Vijay Menon and Keshav Pingali, Cornell University

Using Java Reflection to Automate Extension Language Parsing67
Dale Parson, Bell Laboratories, Lucent Technologies

Monday, October 4

DSLs and Monads

Session Chair: Paul Hudak, Yale University

DSL Implementation Using Staging and Monads81
Tim Sheard, Zine-el-abidine Benaissa, and Emir Pasalic, Oregon Graduate Institute

Monadic Robotics95
John Peterson, Yale University, and Greg Hager, The Johns Hopkins University

Embedded Languages

Session Chair: Michael Schwartzbach, University of Aarhus

Domain-Specific Embedded Compilers109
Daan Leijen and Erik Meijer, University of Utrecht

Verischemelog: Verilog Embedded in Scheme	123
<i>James Jennings and Eric Beuscher, Tulane University</i>	

Tuesday, October 5

The Web, Data, and Collaboration

Session Chair: Jay Lepreau, University of Utah

Declarative Specification of Data-Intensive Web Sites	135
<i>Mary Fernández and Dan Suciu, AT&T Labs—Research; Igor Tatarinov, North Dakota State University</i>	

A Collaboration Specification Language	149
<i>Du Li and Richard R. Muntz, University of California, Los Angeles</i>	

Hancock: A Language for Processing Very Large-Scale Data	163
<i>Dan Bonachea, University of California, Berkeley; Kathleen Fisher and Anne Rogers, AT&T Labs—Shannon Laboratory; Frederick Smith, Cornell University</i>	

Index of Authors

Ardis, Mark A.	15	Muntz, Richard R.	149
Benaissa, Zine-el-abidine	81	Nakatani, Lloyd H.	15
Bershad, Brian N.	1	Olsen, Robert G.	15
Beuscher, Eric	123	Parson, Dale	67
Bonachea, Dan	163	Pasalic, Emir	81
Burnett, Margaret	25	Peterson, John	95
Fernández, Mary	135	Pingali, Keshav	53
Fisher, Kathleen	163	Pontrelli, Paul M.	15
Guyer, Samuel Z.	39	Reichwein, James	25
Hager, Greg	95	Rogers, Anne	163
Jennings, James	123	Rothermel, Gregg	25
Leijen, Daan	109	Sheard, Tim	81
Li, Du	149	Sirer, Emin Gün	1
Lin, Calvin	39	Smith, Frederick	163
Meijer, Erik	109	Suciu, Dan	135
Menon, Vijay	53		

Message from the Program Chair

It is my pleasure to present to you the proceedings of the Second Conference on Domain-Specific Languages (DSL '99), sponsored by USENIX in cooperation with the ACM Special Interest Groups on Programming Languages and Software Engineering.

Domain-specific languages (DSLs) have had a substantial impact on how software is created, maintained, and modified. Prototypical examples of DSLs are YACC, SQL, spreadsheets, and HTML. These DSLs exemplify many of the unique attributes of the DSL approach:

- DSLs automatically provide programmers with guarantees of correctness, performance, and security that are simply unachievable with general-purpose languages such as C, C++, or Java (think of YACC and SQL).
- DSLs allow non-programmers to program (think of spreadsheets).
- By providing high-level abstractions tailored to the problem domain, DSLs allow programmers with general skills to program in a new domain without having to know platform details (think of HTML and Web services).

DSL '99 advances the practice of DSL design, DSL implementation, and software engineering by:

- providing examples of successful DSLs;
- highlighting the spectrum of benefits DSLs provide (e.g., compile-time guarantees of behavior, improved program performance);
- uncovering design principles and methodologies for creating DSLs;
- explicating design techniques and tools for working with DSLs throughout the software engineering life-cycle;
- providing a framework within which language designers from different domains can easily communicate;
- and facilitating a community that will continue to study and refine the practice of software engineering through DSLs.

The program committee and a number of outside reviewers reviewed the 32 papers submitted and selected the 13 papers presented here. While the number of papers submitted was lower than expected (in comparison, DSL '97 received 55 submissions and selected 23 papers), the program committee maintained a high bar for acceptance to insure a top-quality program.

In addition to the technical track papers presented here, the DSL '99 conference had three invited talks and two "hot research reviews." Brad Myers from the Human-Computer Interaction Institute (CMU) gave the keynote address, on the results of empirical studies designed to discover the most natural programming paradigms for non-professional programmers. Peter Lee, from Carnegie Mellon University and Cedilla Systems Inc., spoke on "Language Technology for Performance and Security," and Philip Wadler, from Bell Labs, spoke on the "The Next 700 Markup Languages." The hot research reviews examined DSLs for programming active networks (Carl Gunter, University of Pennsylvania) and explored how to design and create DSLs using program specialization (Charles Consel, Irisa/University of Rennes).

Sincerely,

Thomas Ball

other automated techniques, such as comparative evaluation, can achieve high code coverage. Finally, we show that the structured nature of production grammars can be used to reason about the behavior of test cases, and address the oracle problem. Testing, especially of commercial systems, is a field where success is fundamentally difficult to quantify because the number of undiscovered bugs cannot be known. We relate anecdotal descriptions of the types of errors uncovered by our approach, and use quantitative measures from software engineering, such as code coverage, whenever possible.

In the next section, we provide the necessary background on the Java virtual machine and define the scope and goals of our testing efforts. Readers with a background in the Java virtual machine can skip ahead to the end of that section. Section 3 describes the *lava* language, and illustrates how the language can be used to construct complex test cases from a grammar. Section 4 discusses the integration of production grammars with other automated testing techniques. Section 5 describes extensions to the *lava* language to concurrently generate certificates along with test cases. Section 6 discusses related work, and section 7 concludes.

2. Background & Goals

A Java virtual machine (JVM) is an extensive system combining operating system services and a language runtime around a typed, stack-based, object-oriented instruction set architecture. The integrity of a Java virtual machine relies critically on the three fundamental components that comprise the JVM; namely, on a verifier, interpreter/compiler and a set of standard system libraries. We focus our testing efforts on the first two components. The system libraries are written mostly in Java, and consequently benefit directly from increased assurance in the verifier and the execution engine. Verification and compilation form the bulk of the trusted computing base for a JVM, and embody a large amount of complex and subtle functionality.

The Java verifier ensures that untrusted code, presented to the virtual machine in the richly annotated bytecode format, conforms to an extensive suite of system security constraints. While these constraints are not formally specified, we extracted roughly 620 distinct security axioms through a close examination of the Java virtual machine specification and its errata [Lindholm&Yellin 99]. These safety axioms range in complexity from integer comparisons to sophisticated analyses that rely on type inference and data flow. Roughly, a third of the safety axioms are devoted to

ensuring the consistency of top level structures within an object file. They deal with issues such as bound checks on structure lengths and range checks on table indices. Around 10% of the safety checks in the verifier are devoted to ensuring that the code section within the object file is well structured. For instance, these checks restrict control flow instructions from jumping outside the code segment or into the middle of instructions. Another 10% of the checks are devoted to making sure that the assumptions made in one class about another class are valid. These checks ensure, for example, that a field access made by one class refers to a valid exported field in another class of the right type. The remaining 300 or so checks are devoted to dataflow analysis, and form the crux of the verifier. They ensure that the operands for virtual machine instructions will be valid on all dynamic instruction paths. Since Java bytecode does not contain declarative type information, the type of each operand has to be inferred for all possible execution paths. This stage relies on standard dataflow analysis to exhaustively check the system state on all possible paths of execution, and ensures, for instance, that the operands to an integer addition operation always consist of two integers. Further, these checks also ensure that objects are properly initialized via parent constructors, and that every use of an object is preceded by an initialization of the same object. Based on various analyses [Drossopoulou+ 97,Drossopoulou+ 99,Syme 97], we assume that a correct implementation of these 620 axioms is sufficient to ensure typesafety and to protect the assumptions made in the virtual machine from being violated at runtime. The task of the system designer, then, is to ensure that the implementation of safety axioms in a verifier corresponds to the specification.

Similarly, the Java execution engine, whether it is an interpreter, a just-in-time compiler, a way-ahead-of-time compiler [Proebsting et al. 97] or a hybrid scheme [Muller et al. 97,Griswold], needs to implement a large amount of functionality to correctly execute Java bytecode instructions. Since the JVM instruction set architecture defines a CISC, the instruction semantics are high-level and complex. There are 202 distinct instructions, whose behavioral description takes up more than 160 pages in the JVM specification. In addition, the architecture allows non-uniform constructs such as arbitrary length instructions, instruction padding and variable opcode lengths, which introduce complications into the implementation.

The challenge for testing JVMs, then, is to ensure that the implementation of safety checks in a bytecode verifier, and that the implementation of instruction

semantics in an interpreter, compiler and optimizer, are correct. We faced this challenge when we developed our own Java virtual machine [Sirer et al. 98]. Our attempts to create test cases manually were soon overwhelmed, and we sought a testing scheme that possessed the following properties:

- *Automatic*: Testing should proceed without human involvement, and therefore be relatively cheap. The technique should be easy to incorporate into nightly regression testing.
- *Complete*: Testing should generate numerous test cases that cover as much of the functionality of a virtual machine as possible. It should also admit a metric of progress that correlates with the amount of assurance in the virtual machine being tested.
- *Conservative*: Bad Java bytecodes should not be allowed to pass undetected through the bytecode verifier, and incorrectly executed instructions in the compiler or interpreter should be detected.
- *Well structured*: Examining, directing, checkpointing and resuming verification efforts should be simple. Error messages should be descriptive; that is, it should be easy for a programmer to track down and fix a problem.
- *Efficient*: Testing should result in a high-confidence Java virtual machine within a reasonable amount of time.

The rest of this paper describes our experience with *lava* aimed at achieving these goals.

3. *Lava* and Grammar-based Test Generation

Our approach to test generation is to use an automated, well-structured process driven by a production grammar. A production grammar is the opposite of a regular parsing grammar in that it produces a program (i.e. all terminals, or tokens) starting from a high-level description (i.e. a set of non-terminals). The composition of the generated program reflects the restrictions placed on it by the production grammar. Figure 1 illustrates the high-level structure of the test generation process. A generic code-generator-generator parses a Java bytecode grammar written in *lava* and emits a specialized code-generator. The code-generator is a state machine that in turn takes a seed as input and applies the grammar to it. The seed consists of the high-level description that guides the production process. Running the code-generator on a seed produces

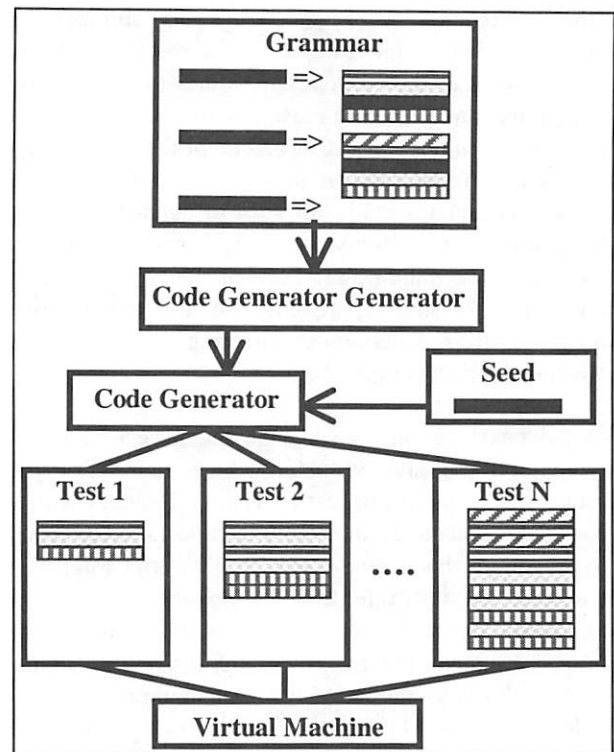


Figure 1. The structure of the test generation process. A code-generator-generator parses a production grammar, generates a code-generator, which in turn probabilistically generates test cases based on a seed.

test cases in Java bytecode that can then be used for testing.

The input to the *lava* code-generator-generator consists of a conventional grammar description that resembles the LALR grammar specification used in *yacc* (Figure 2). This context-free grammar consists of productions with a left-hand side (LHS) containing a single non-terminal that is matched against the input. If a match is found, the right-hand side (RHS) of the production replaces the LHS. As with traditional parser systems, we use a two-phase approach in *lava* for increased efficiency and ease of implementation. The code-generator-generator converts the grammar specification into a set of action tables, and generates a code-generator that performs the actual code production based on a given seed. In essence, the code-generator-generator performs code specialization on a generic production system for a particular grammar. We picked this two-phase approach to increase the efficiency of the code-generator through program specialization.

Using Production Grammars in Software Testing

Emin Gün Sirer

Brian N. Bershad

*Department of Computer Science
University of Washington
Box 352350, Seattle, WA 98195-2350
<http://kimera.cs.washington.edu>
{egs,bershad}@cs.washington.edu*

Abstract

Extensible typesafe systems, such as Java, rely critically on a large and complex software base for their overall protection and integrity, and are therefore difficult to test and verify. Traditional testing techniques, such as manual test generation and formal verification, are too time consuming, expensive, and imprecise, or work only on abstract models of the implementation and are too simplistic. Consequently, commercial virtual machines deployed so far have exhibited numerous bugs and security holes.

In this paper, we discuss our experience with using production grammars in testing large, complex and safety-critical software systems. Specifically, we describe *lava*, a domain specific language we have developed for specifying production grammars, and relate our experience with using *lava* to generate effective test suites for the Java virtual machine. We demonstrate the effectiveness of production grammars in generating complex test cases that can, when combined with comparative and variant testing techniques, achieve high code and value coverage. We also describe an extension to production grammars that enables concurrent generation of certificates for test cases. A certificate is a behavioral description that specifies the intended outcome of the generated test case, and therefore acts as an oracle by which the correctness of the tested system can be evaluated in isolation. We report the results of applying these testing techniques to commercial Java implementations. We conclude that the use of production grammars in combination with other automated testing techniques is a powerful and effective method for testing software systems, and is enabled by a special purpose language for specifying extended production grammars.

1. Introduction

This paper describes *lava*, a special purpose language for specifying production grammars, and summarizes

how production grammars can be used as part of a concerted software engineering effort to test large systems. In particular, we describe our experience with applying production grammars written in *lava* to the testing of Java virtual machines. We show that production grammars, expressed in a suitable language, can be used to automatically create, and reason about, complex test cases from concise, well-structured specifications.

Modern virtual machines [Lindholm&Yellin 99, Inferno, Adl-Tabatabai et al. 96], such as Java, have emerged in recent years as generic and ubiquitous components in extensible applications. Virtual machines can now be found in hypertext systems [Barners-Lee et al. 96], web servers [SunJWS], databases [Oracle], desktop applications, and consumer devices such as cellphones and smartcards. The primary appeal of virtual machines is that they enable users to safely run untrusted, and potentially malicious, code. The safety of modern virtual machines, Java in particular, depends critically on three large and complex software components: (1) a verifier for static inspection of untrusted code against a set of safety axioms, (2) an interpreter or a compiler to respect instruction semantics during execution, and (3) a runtime system to correctly provide services such as threading and garbage collection. A failure in any one of these components allows applications to violate system integrity, and may result in data theft or corruption [McGraw & Felten 96].

This capability to execute untrusted code places a large burden on the system implementers to ensure overall system safety and correctness. A typical modern virtual machine such as Java contains hundreds of fine-grain, subtle and diverse security axioms in its verifier, as well as many lines of similarly subtle code in its interpreter, compiler and system libraries. Consequently, verifying the correctness of a virtual machine is a difficult task.

Testing Alternatives

The current state of the art in formal program verification is not yet sufficient to verify large, complex virtual machines. While the theory community has made substantial progress on provably correct bytecode verifiers [Stata&Abadi 98, Freund&Mitchell 98], current results exhibit three serious shortcomings. Namely, they operate on abstract models of the code instead of the actual implementation, cover only a subset of the constructs found in a typical virtual machine and require extensive human involvement to map the formal model onto the actual implementation. In addition, these type-system based formal methods for verifiers do not immediately apply to interpreters, compilers or the runtime system, whose functionality is hard to model using static type systems.

Subsequently, virtual machine developers have had to rely on manual verification and testing techniques to gain assurance in their implementations. Techniques such as independent code reviews [Dean et al. 97] and directed test case generation have been used extensively by various commercial virtual machine vendors. However, manual testing techniques, in general, are expensive and slow due to the amount of human effort they entail. In addition, independent code reviews require relinquishing source code access to reviewers, and lack a good progress metric for the reviewers' efforts. Manual test case generation, on the other hand, often requires massive amounts of tedious human effort to achieve high code coverage.

A common approach to automating testing is to simply use ad hoc scripts, written in a general-purpose language, to generate test inputs. While this approach may save some time in test creation, it exhibits a number of shortcomings. First, writing such test scripts is time consuming and difficult, as scripting languages often do not provide any convenient constructs for this programming domain beyond string manipulation. Second, managing many such scripts is operationally difficult once they have been written, especially as they evolve throughout the life cycle of the project. Third, the unstructured nature of general-purpose languages poses a steep learning curve for those who need to understand and modify the test scripts. Fundamentally, a general-purpose language is too general, and therefore too unstructured, for test generation.

Production Grammars

In this paper, we describe *lava*, a special purpose language for specifying production grammars. A produc-

tion grammar is a collection of non-terminal to terminal mappings that resembles a regular parsing grammar, but is used "in reverse." That is, instead of parsing a sequence of tokens into higher level constructs, a production grammar generates a stream of tokens from a set of non-terminals that specify the overall structure of the stream. We describe how *lava* grammars differ from traditional parsing grammars, and illustrate the different features we added into the language to support test generation. Our experience with *lava* demonstrates that a special purpose language for specifying production grammars can bring high coverage, simplicity, manageability and structure to the testing effort.

Production grammars are well-suited for test generation not only because they can create diverse test cases effectively, but also because they can provide guidance on how the test cases they generate ought to behave. It is often hard to determine the correct system behavior for automatically generated test cases -- a fundamental difficulty known as the oracle problem. In the worst case, automated test generation may require reverse engineering and manual examination to determine the expected behavior of the system on the given test input.

We address the oracle problem in two ways. We first show that test cases generated with production grammars can be used in conjunction with comparative testing to create effective test suites without human involvement. Second, we show how an extended production grammar language can generate self-describing test cases that obviate the need for comparative testing. In simple production systems, the code producer does not retain or carry forward any information through the generation process to be able to reason about the properties of the test case. We overcome this problem by extending the grammar specification language to concurrently generate certificates for test cases. Certificates are a concise description of the expected system behavior for the given test case. They act as oracles on the intended behavior of the generated test program, and thereby enable a single virtual machine to be tested without comparison against a reference implementation.

Overall, this paper makes three contributions. First, we describe the design of a domain specific language for specifying production grammars, and illustrate how it can be used on its own for carrying out performance analyses, for checking robustness, and for testing transformation components such as compilers. Further, we show that combining production grammars with

Within the code-generator, the main data structure is a newline-separated stream, initially set to correspond to the seed input. Being able to specify arbitrary seeds enables us to modify parts of existing programs in

```
Grammar-Rule := name [limit]
              identifier ">" identifier*
              [guard] [action]
Identifier := Token | Variable
Token := [a-zA-Z_][a-zA-Z0-9_]*
Variable := "$" Token
Guard := "guard{" code "}"
Action := "action{" code "}"
```

Figure 2. The grammar for the *lava* input language in EBNF form. Its simplicity makes the language easy to adopt, while variable substitution and arbitrary code sequences make it powerful.

controlled ways. Each line in this stream is scanned, and occurrences of an LHS are replaced by the corresponding RHS. If there is more than one possible action for a given match, the code-generator picks an outcome probabilistically, based on weights that can be associated with each production. When all possible non-terminals are exhausted, the code-generator outputs the resulting stream. We then use the Jasmin bytecode assembler [Meyer & Downing 97] to go from the textual representation to the binary bytecode format.

Lava grammars can be annotated with three properties that are useful for test generation. First, each production rule can have an associated name. Along with each test case, the code-generator creates a summary file listing the names of the grammar rules that gave rise to that test case, thereby simplifying test selection and analysis. We used these summary files extensively during JVM development to pick (or to skip) test cases that exhibit certain instruction sequences, as the high-level descriptions were easier to act on than reverse engineering and pattern-matching instruction sequences. In addition to a name, each grammar rule in *lava* may have an associated limit on how many times it can be exercised. Java, unlike most language systems, enforces certain structural limits, such as limits on code length per method, in addition to any limits the tester may impose to guide the test generation process. The limits on production rules enable the grammar to accommodate such constraints. Finally, in order to enable the production of context-sensitive outputs, *lava* allows an optional code fragment, called an action, to be associated with each production. This code is executed when the production is selected during code-generation, in a manner analogous to context-sensitive parsing using *yacc* grammars. An escape

sequence facilitates the substitution of variables defined in actions on the RHS of productions. Actions are widely used in this manner to generate labels, compute branch targets, and perform assignment of values to unique local variables. The code in action blocks, like the entire *lava* system, is written in AWK [Aho et al. 88].

In addition to these attributes, each production carries a weight that determines which productions fire in case more than one are eligible for a given LHS. This case occurs frequently in code production, as it is quite often possible to insert just about any instruction se-

```
insts 5000 INSTS => INST INSTS
emptyinst INSTS => /* empty */
ifeqstmt
INST => iconst_0; INSTS; ifeq L$1;
      jmp L$2; L$1: INSTS; L$2:
jsrstmt
INST => jsr L$1; jmp L$2;
      L$1: astore $reg; INSTS;
      ret $reg; L$2:
action{ ++reg; }
```

```
Weight jsrstmt 10
Weight ifeqstmt 10
Weight insts 1
Weight emptyinst 0

.class public testapplet$NUMBER
.method test()V
    INSTS; return
.end method
```

Figure 3. A simplified *lava* grammar and a corresponding seed for its use. Non-terminals are in upper case and dollar signs indicate that the value of the variable should be substituted at that location. Weights in the seed specify the relative mix of *jsr* and *if* statements. Overall, the grammar specification is concise and well-structured.

quence at any point within a program. The probabilistic selection based on weights, then, introduces non-determinism and enables each code-generator run to produce a different test case. The weights, specified as part of the seed input, determine the relative occurrences of the constructs that appear in the grammar. Separating the weights from the rest of the grammar enabled us to generate different instruction mixes without having to rewrite or rearrange the grammar specification.

Figure 3 shows a sample grammar written in *lava*. This particular grammar is intended to exercise various control flow instructions in Java virtual machines, and is a simplified excerpt from a larger grammar that covers all possible outcomes for all branch instructions in bytecode. The **insts** production has a specified limit of 5000 invocations, which restricts the size of the generated test case. The two main productions, **jsrstmt** and **ifegstmt**, generate instruction sequences that perform subroutine calls and integer equality tests. These concise descriptions, when exercised on the seed shown, generate a valid class file with complicated branching behavior. The seed itself is a skeletal class file whose code segment is to be filled in by the grammar. Equal weighting between the **if** and the **jsr** statements ensures that they are represented equally in the test cases. A weight of 0 for the **emptyinst** production effectively disables this production until the limit on the **insts** production is reached, ensuring that test generation does not terminate early. The grammar also illustrates the use of unique labels and the integration of actions with the grammar in order to create context-sensitive code. While the *lava* input language is quite general and admits arbitrarily complex grammar specification, we found that most of the grammars we wrote used common idioms which reflected the constructs we wished to test, as in the example above. Figure 4 contains a sample output generated by this toy grammar that was designed to generate “spaghetti code.” Overall, the description language is concise, the test generation process is simple, and the generated tests are complex.

iconst_0	L0: jsr L6	L14:
iconst_0	jmp L7	L15:ret 0
jsr L18	L6: astore 1	L3: iconst_0
jmp L19	ret 1	ifeq L8
L18:astore 3	L7: jsr L12	jmp L9
ret 3	jmp L13	L8:
L19:ifeq L4	L12:astore 2	L9: iconst_0
jmp L5	ret 2	ifeq L16
L4:	L13:	jmp L17
L5: iconst_0	L1: jsr L2	L16:
ifeq L10	jmp L3	return
jmp L11	L2: astore 0	
L10:	iconst_0	
L11:ifeq L0	ifeq L14	
jmp L1	jmp L15	

Figure 4. A sample method body produced by the grammar shown in Figure 3. The resulting test cases are complex, take very little time to produce, and are more reliable than tests written by hand.

Initially, we considered *lava*’s limitation of left-hand sides to a single non-terminal as a serious restriction that would need to be addressed. We found, however, that cases that warrant multiple non-terminals on the

LHS arise rarely in practice. In contrast, we found that there were numerous cases where a context free grammar would indicate that a given production is eligible for a particular location, but context-sensitive information prohibited such an instruction from appearing there. For example, the use of an object is legal only following proper allocation and initialization of that object. While it is possible to keep track of object initialization state by introducing new non-terminals and rules into the grammar, this overly complicates grammar specifications, and scales badly with the number of objects to be tracked. It is significantly simpler to keep track of such state in auxiliary data structures through actions, much like in *yacc* grammars. However, since these data structures are not visible to the code production engine, the code-generator may lack the information it needs to decide for which spots certain productions are not eligible. To address this problem, we introduced guards into the grammar specification. A guard is a Boolean function that is consulted to see if a given production can be used in a particular context. If the guard returns false, that replacement is not performed, and another production is selected probabilistically from the remaining set eligible for that substitution. Hence, guards enable the context-sensitive state that is tracked through actions to influence the choice of productions. While it is possible to use guards to associate probabilistic weights with productions, we decided to retain explicit support for weights in the language. Weights are such a common idiom that special-casing their specification both greatly simplifies the grammars, and increases the performance of the code generator. Further, using guards to implement weights would encode probabilities in the grammar specification, whereas in our implementation, they are specified on a per-seed basis, obviating the need for testers to modify the grammar specifications.

Results

We have used *lava* in a number of different testing scenarios including performance analysis and fault detection of the virtual machine, and integrity of code transformation engines such as compilers and binary rewriters.

First, we used the test cases to test for easy-to-detect errors, such as system crashes. Typesafe systems such as virtual machines are never supposed to crash on any input. A crash indicates that typesafety was grossly violated, and that the error was luckily caught by the hardware protection mechanisms underneath the virtual machine. We used a simple grammar with three

productions, and the very first test case that we generated found that the Sun JDK1.0.2 verifier on the DEC Alpha would dump core when faced with numerous deeply nested subroutine calls.

Second, we used stylized test cases generated by *lava* for characterizing the time complexity of our verifier as a function of basic block size and total code length. The parameterizable nature of the grammar facilitated test case construction, and only required a few runs of the code-generator with different weights and seeds. It took less than 10 minutes to write the grammar from scratch, and only several minutes to generate six test cases ranging in size from a few instructions to 60000. Each generated test case was unique, yet was comparable to the other tests on code metrics such as average basic block size and instruction usage. These tests uncovered that our verifier took time $O(N^2 \log N)$ in the number of basic blocks in a class file, and motivated us to find and fix the unnecessary use of ordered lists where unordered ones would have sufficed.

Finally, we used the generated tests to verify the correctness of Java components that perform transformations. Components such as Java compilers and binary rewriting engines [Sirer et al. 98] must retain program semantics through bytecode to native code or bytecode to bytecode transformations. Such components can be tested by submitting complex test cases to the transformations, and comparing the behavior of the transformed, e.g. compiled, program to the original. In our case, we wanted to ensure that our binary rewriter, which performs code movement and relocation, preserved the original program semantics through its basic block reordering routine. Our binary rewriter is a fairly mature piece of code, and we had not discovered any bugs in it for a while. To test the rewriter, we created 17 test cases which averaged 1900 instructions in length and exercised every control flow instruction in Java. The grammar was constructed to create test cases with both forward and backward branches for each type of jump instruction, and instrumented to print a unique number in each basic block. We then simply executed the original test cases, captured their output and compared it to the output of the test cases after they had been transformed through the binary rewriter. This testing technique caught a sign extension error that incorrectly calculated backward jumps in **tableswitch** and **lookupswitch** statements, and took a total of two days from start to finish.

Though production grammars facilitate test case generation, they do not by themselves provide a complete solution to software testing. The main problem in

software testing has to do with determining when the tested system is behaving correctly on a given input, and when it is not. In the next section, we illustrate how this problem can be addressed by comparative evaluation.

4. Comparative Testing with Variations

Our first approach to addressing the oracle problem, that is, the problem of detecting when a virtual machine is misbehaving on a given input, is to rely on multiple implementations of a virtual machine to act as references for each other in a process commonly known as comparative, or differential, testing. The core idea in comparative testing is simply to direct the same test cases to two or more versions of a virtual machine, and to compare their outputs. A discrepancy indicates that at least one of the virtual machines differs from the others, and typically requires human involvement to determine the cause and severity of the discrepancy. Since the Java virtual machine interface is a de facto standard for which multiple implementations are available, we decided to use it to address the oracle problem for automatically generated test cases.

A separate reason compelled us to expand comparative testing by introducing variations into the test cases generated by production grammars. A variation is simply a random modification of the test case to generate a new test. The assembly language in which the test cases were generated hide some of the details of the Java bytecode format, making it impossible to exercise

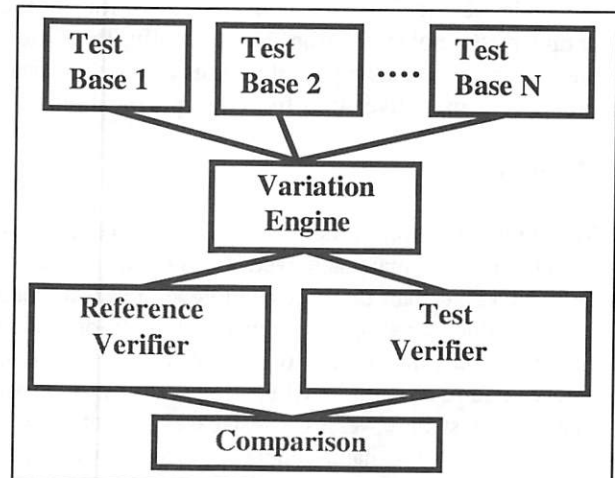


Figure 5. Comparative Evaluation. A variation engine injects errors into a set of test bases, which are fed to two different bytecode verifiers. A discrepancy indicates an error, a diversion from the specification, or an ambiguity in the specification.

certain conditions. For example, there is no way to generate a class file which exhibits internal consistency errors, such as out of bounds indices, using an assembly language such as Jasmin. To overcome this difficulty of manipulating low-level constructs from assembly language, we decided to introduce variations into the tests generated by *lava*. The overall process we followed is illustrated in Figure 5.

We compared our virtual machine implementation to Sun JDK 1.0.2 and the Microsoft Java virtual machine found in Internet Explorer 4.0. We formed the test inputs by taking the output of a production grammar and introducing one-byte pseudo-random modifications, or variations. We experimented with introducing multiple modifications into a single test base, but found that multiple changes made post-mortem investigation of discrepancies more difficult. Therefore, each test case was generated by introducing only a single one-byte value change at a random offset in a base class file.

In most cases, all consulted bytecode verifiers agreed that the mutated test inputs were safe or unsafe, yielding no information about their relative integrity. In some cases, however, one of the bytecode verifiers would disagree from the rest. Cases where the class file was accepted by our bytecode verifier but rejected by a commercial one usually pointed to an error in our implementation. We would fix the bug and continue. Conversely, cases where the class file was rejected by our bytecode verifier but accepted by a commercial bytecode verifier usually indicated an error in the commercial implementation. Occasionally, the differences were attributable to ambiguities in the specification or to benign diversions from the specification.

Results

We found that the complex test cases generated by production grammars achieved as good as or better code coverage than the best hand-generated tests, had higher value coverage, and were in addition much easier to construct. Figure 6 examines the number of safety axioms in our verifier triggered by a hand-generated test base versus a test base generated by a *lava* grammar. The hand-generated test case was laboriously coded to exercise as many of the features of the Java virtual machine, in as diverse a manner as possible. The graph shows that automatically generated test cases exercise more check instances than hand-generated test cases. Indeed, after 30000 variations, the coverage attained by automatically generated test cases is a strict superset of the manually generated

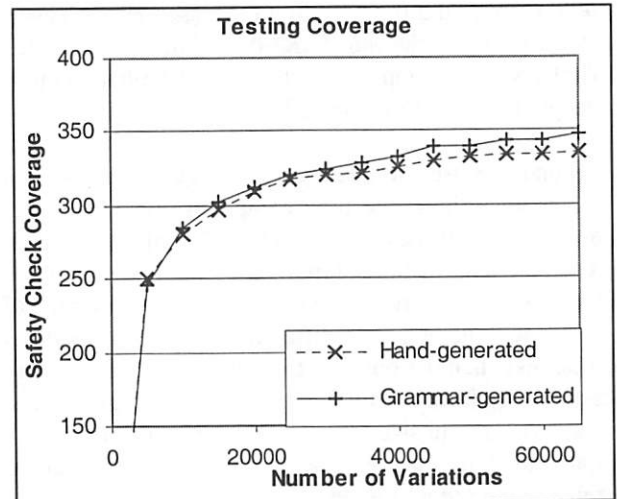


Figure 6. A plot of code coverage for hand generated and *lava* generated test cases shows that automatically generated test cases are as effective as carefully constructed hand-generated test cases at achieving breadth of coverage.

tests. We attribute this to the greater amount of complexity embodied in the test cases generated by the grammar. Further, the tests that are exercised only by grammar-based tests are those cases, such as polymorphic merges between various types at subroutine call sites, for which manual tests are especially hard and tedious to construct.

Further, we found that comparative testing with variations is fast. At a throughput of 250K bytes per second on a 300 MHz Alpha workstation with 128K of memory on five different base cases, comparative evaluation with variations exercised 75% of the checks in the bytecode verifier within an hour and 81% within a day. Since our test bases consisted of single classes, we did not expect to trigger any link-time checks, which accounted for 10% of the missing checks in the bytecode verifier. Further inspection revealed that another 7% were due to redundant checks in the bytecode verifier implementation. More specifically, there were some checks that would never be triggered if other checks were implemented correctly. For example, long and double operands in Java take up two machine words, and the halves of such an operand are intended to be indivisible. Although various checks in the bytecode verifier prohibit the creation of such operand fragments, our bytecode verifier redundantly checks for illegally split operand on every stack read for robustness.

We attribute this high rate of coverage to three factors. First, Java bytecode representation is reasonably com-

pact, such that small changes often drastically alter the semantics of a program. In particular, constant values, types, field, method and class names are specified through a single index into a constant pool. Consequently, a small variation in the index can alter the meaning of the program in interesting ways, for instance, by substituting different types or values into an expression. For example, one of the first major security flaws we discovered in commercial JVM implementations stemmed from lack of type-checking on constant field initializers. A single variation, within the first 10000 iterations (15 minutes) of testing, uncovered this loophole by which integers could be turned into object references.

A second reason that one-byte variations achieve broad coverage is that the Java class file format is arranged around byte-boundaries. Since the variation granularity matches the granularity of fields in the class file format, a single change affects only a single data structure at a time, and thereby avoids wreaking havoc with the internal consistency of a class file. For example, one test uncovered a case in both commercial bytecode verifiers where the number of actual arguments in a method invocation could exceed the declared maximum. Had this field been packed, the chances of a random variation creating an interesting case would have been reduced. Similarly, random variations uncovered a case in the MS JVM where some branches could jump out of code boundaries, likely due to a sign extension error. Packing the destination offset would have made it harder for single byte variations to locate such flaws.

Finally, we attribute the effectiveness of this approach to the complexity of the test bases generated by *lava* grammars. In testing with one-byte variations, the total complexity of the test cases is limited primarily by the original complexity of the test bases. The grammar based test cases achieve high-complexity with very little human effort.

While comparative testing with variations partially addresses the oracle problem and enables the complex test cases generated by production grammars to be used effectively in testing, it requires another implementation of the same functionality to work. Even though finding alternative implementations was not at all difficult in the case of the Java virtual machine, comparative evaluation is not always the best testing approach. A duplicate may not be available or different implementations may exhibit too many benign differences. Most importantly, even when there is a second implementation to compare against, it may be

in error in exactly the same way as the implementation of interest.

5. Producing Self-Describing Test Cases

We address the shortcomings of comparative evaluation by extending grammar testing to generate certificates concurrently with test cases. A certificate is a behavioral description that specifies the intended outcome of the generated test case, and therefore acts as an oracle by which the correctness of the tested system can be evaluated in isolation. The insight behind this technique is that certificates of code properties allow us to capture both static and dynamic properties of test programs, such as their safety, side effects or values they compute. The behavior of a virtual machine can then be compared against the certificate to check that the virtual machine is implemented correctly.

A Sample *Lava* Grammar and Corresponding Behavioral Description

```

1: STMTS => STMT STMTS
2: STMTS => nil
   : N = λt.t
3: STMT => iconst 0
   : A = λS.λt.(S (cons 0 t))
4: STMT => iconst 1
   : B = λS.λt.(S (cons 1 t))
5: STMT => iadd
   : C = λS.λt.(S (cons
                     (+ (car t) (cadr t)) (cddr t)))
6: STMT => ifeq LN+1;
   LN: STMTS; goto LN+2
   LN+1: STMTS
   LN+2: STMTS
   : D = λS1.λS2.λS3.λt.(if
                           (equal (car t) 0)
                           (S3 (S1 (cdr t)))
                           (S3 (S2 (cdr t)))))

```

Figure 7. A sample grammar with a corresponding behavioral description.

Two types of useful certificates may accompany synthetically generated code. The first form of a certificate is a proof over the grammar, which can accompany all test programs generated by that specification as a guarantee that they possess certain properties. For instance, it is easy to see, and inductively prove, that the sample grammar given in Figure 3 will execute all labeled blocks it generates. Using a third production,

we can instrument every labeled basic block produced by that grammar, and check to ensure that a virtual machine enters all of the blocks during execution. We simply print the block label when the block is entered, and, at the end of the test run, check to make sure that all blocks appear exactly once in the trace. Using this technique, for instance, we found that some of the control flow instructions in an early version of our interpreter had the sense of their comparisons reversed.

Not all grammars lend themselves to such inductive proofs. We found a second form of certificates, describing the runtime behavior of a specific test, often more applicable. In essence, what the tester needs to overcome the oracle problem is a specification of what the test ought to compute, in a format other than Java bytecode that can be automatically evaluated. We chose to use lambda expressions for specifying such certificates because of their simplicity. Such certificates are difficult to construct by hand, however, as they require reasoning about the composition of the productions that generated that test case. Instead, we generate these certificates concurrently with the test program, based on annotations in the grammar. The annotations, in lambda expression form, are arranged by the programmer such that they capture interesting aspects of corresponding productions. The *lava* system merges these annotations together through the test production process, and generates a resulting lambda expression, which computes the same result as the test case when evaluated. The correctness of a compiler or interpreter can then be checked by executing the test case and comparing with the result of the lambda expression.

We have extended *lava* such that it takes two grammars, one for the intended test language and one for the behavioral description. Whenever a production in the first grammar is chosen, the corresponding production rule in the second grammar is also selected. Test generation thus creates two separate outputs corresponding to the test case and its behavioral description. We have used Scheme for our behavioral description language, and thus can compare the correctness of a Java virtual machine simply by evaluating a test program in Java and comparing its output to that of the Scheme program.

Figure 7 illustrates a behavioral specification for a limited grammar that supports stack push, arithmetic and control flow operations. Our behavioral descriptions use lambda expressions in continuation passing style. Each production is annotated with an expression that takes the current machine state and continuation

as input, and performs the operations of the right hand side. Non-terminals appearing on the RHS constitute free variables, and each production step forms a substitution. The overall behavioral description is formed by substituting lambda expressions corresponding to productions for free variables. For instance, suppose that we start with a seed of "STMTS" and pick productions 1, 3, 1, 4, 1, 5, 1, 6, 1, 4, 2, 1, 3, 2, 2. The corresponding behavioral description would be (A (B (C (((D (B N)) (A N)) N))))). This expression, when evaluated on the initial state, yields (1), the result of executing the program. If execution of the generated bytecode disagrees with this result, there is a fault with either the virtual machine implementation or the lambda expression evaluator. Consequently, this approach enables checking the correctness of a JVM against a simple, formally-analyzed and time-tested implementation.

While behavioral descriptions in *lava* resemble operational semantics, they are vastly simpler to specify than a full formal semantics for the Java virtual machine. This is because the behavioral descriptions need to be provided not for every construct in the language, but only for idioms of interest, and need capture only those properties of the idioms in which the tester is interested. For example, providing the correct semantics for each of the Java virtual machine's control flow opcodes, which include conditional branches, jumps, method invocations, subroutine calls and exceptions, is a difficult task [Drossopoulou+ 98]. However, effects of an idiom, say one that consists of a jump, loop, exception raise, compute and return from exception, which uses these opcodes to push a predetermined value onto the stack, is easy to capture in a behavioral description. Essentially, *lava* enables the tester to provide partial operational descriptions at the production level, instead of having to provide complete and

Currently, *lava*'s support for extended grammars is not as general as we would like. The behavioral descriptions are limited to grammars where the main production is right-recursive, as shown in production 1 in Figure 6.

Overall, extended code generation with certificates determines precise properties about test cases without recourse to a second implementation of the same functionality. This powerful technique is widely applicable in testing virtual machine components, including bytecode verifiers, compilers, and interpreters.

6. Related Work

Production grammars have been studied from a theoretical perspective, and adopted to a few domains besides testing. [Dershowitz+ 90, Dershowitz 93] give an overview of rewriting systems, of which production grammars are a subclass, and provide a theoretical foundation. In practice, production grammars have been used to describe fractals, and have been adopted in computer graphics to efficiently generate complex visual displays from simple and compact specifications [Prusinkiewicz et al. 88]. Our approach expands this work by applying production grammars to the testing of complex systems, and, within the testing field, overcomes the oracle problem by generating self-describing certificates concurrently with test cases to aid standalone testing.

There has been substantial recent work on formal methods for ensuring correctness of bytecode verifiers, especially on using type-systems as a foundation for bytecode verifier implementations. Stata and Abadi have formally analyzed Java subroutines for type-safety and postulated a type-system that can be used as the basis of a Java verifier [Stata&Abadi 98]. Freund and Mitchell have further extended this model to cover object allocation, initialization and use [Freund&Mitchell 98, Freund&Mitchell 99]. While these approaches are promising and address one of the most safety critical components in a virtual machine, to date, only a subset of the various constructs permitted by the Java virtual machine have been covered by these type system-based frameworks. Further, these approaches operate on abstract models of the code instead of the implementation, and require extensive human involvement to ensure that the model accurately captures the behavior of the bytecode verifier. The strong guarantees of formal reasoning are undermined by the manual mapping of a model onto the actual implementation.

An approach to verification that is common in industry is to perform source code audits, where a group of programmers examine the code and manually check for weaknesses. The Secure Internet Programming group at Princeton has found a number of security flaws in Java implementations using this technique [Dean et al. 97]. The primary shortcoming of manual audits is that they require intensive human labor, and are thus expensive. Further, providing source access to auditors is not always possible or desirable.

Production grammars have been used in conjunction with comparative evaluation to check compiler im-

plementations for compatibility [McKeeman 98]. The author used a stochastic C grammar to generate test cases for C compilers. This work focused primarily on ensuring compatibility between different compilers, and relied on comparative evaluation. This approach resembles our second technique, and suffers from the same problem of requiring a second, preferably stronger implementation of the tested application. We extend this work by generating certificates for test cases that enable the testing of a program without reference to a second implementation. Similarly, [Celenano+ 80] outlines a scheme for generating minimal test cases that cover a BNF grammar, and in their experiences section, mention that the minimal test cases are at times too simple to test interesting inputs. Use of probabilistic productions in our scheme generate much more complex test cases, and provide a point of control for steering the testing effort.

Within the software engineering community, there are various efforts aimed at automatic test generation from formal specifications. [Weyuker et al. 94] describes a set of algorithms for generating tests from boolean specifications. [Mandrioli et al. 95] outline the TRIO system, which facilitates the semi-automatic creation of test cases from temporal logic specifications. [Chang et al. 95] describes a flexible, heuristic-driven, programmable scheme for generating tests from specifications in ADL, a language based on first-order predicate logic. [Gargantini+ 99] illustrates how a model checker can be used to create tests from state machine specifications, and [Bauer&Lamb 79] discusses how to derive test cases from functional requirements for state machines. These efforts all require a formal, modular and accurate specification of the system to be tested in a restrictive formal specification language. Such specifications are not always possible to write for large, complex, and in particular legacy systems, and require expertise to develop and maintain. Further, it is not clear how well the testing process can be steered using these automated testing techniques.

Proof carrying code [Necula 97] and certifying compilers [Necula & Lee 98] resemble extended grammars in that they associate certificates, in their case formal proofs, with code. Proof carrying code associates inherently tamper-proof typesafety proofs with native code, enabling it to be safely integrated into a base system. Certifying compilers are an extension of proof carrying code, where a compiler automatically generates safety proofs along with binary code. The safety proofs are expressed as statements in first-order logic, and are derived and carried forward from the type-

safety properties of the source language. Our work shares the same insight that associating mechanically parseable certificates with code is a powerful technique, but differs from certifying compilers in two ways. First, and most fundamentally, we use extended grammars to ascertain the correctness of a virtual machine implementation, whereas certifying compilers ascertain the safety of a compiled application program. And second, certifying compilers require their input to be written in a typesafe language and construct proofs attesting to static properties such as typesafety, whereas we generate behavioral descriptions based on a programmer specified specification, and reason about dynamic program properties such as program results.

7. Conclusion

In this paper, we have described *lava*, a language for specifying production grammars, and demonstrated how production grammars can be used in software testing. *Lava* grammars enable a well-structured, manageable and simple testing effort to be undertaken using concise test descriptions. The language facilitates the construction of complex test cases, which can then serve as bases for comparative testing. An extended form of the *lava* language can not only generate interesting test cases, but also generate certificates about their behavior. Overall, these techniques enable the verification of large and complex software systems, such as virtual machines, cheaply and effectively.

Systems get safer only if they are designed, developed and refined with testing in mind, and only if the requisite testing infrastructure is in place to guide this effort. We hope that as virtual machines, which have promised safety but not yet delivered, become more pervasive, their growth is accompanied by the adoption of automated verification techniques.

Acknowledgements

We would like to thank Eric Hoffman and the anonymous reviewers for their comments on earlier drafts of this paper. Arthur J. Gregory and Sean McDirmid implemented parts of our Java virtual machine. We would also like to thank Jim Roskind of Netscape, Charles Fitzgerald and Kory Srock of Microsoft, and Tim Lindholm, Sheng Liang, Marianne Mueller and Li Gong of Javasoft for working with us to address the Java security flaws found in commercial verifiers by this work.

References

- [Adl-Tabatabai et al. 96] Adl-Tabatabai, A., Langdale, G., Lucco, S. and Wahbe, R. "Efficient and Language-Independent Mobile Programs." In Conference on Programming Language Design and Implementation, May 1996, p. 127-136.
- [Aho et al. 88] Aho, A.V., Kernighan, B.W., Weinberger P. J. The Awk Programming Language. Addison-Wesley, June 1988.
- [Bauer&Lamb 79] Bauer, J.A. and Finger, A.G. "Test Plan Generation Using Formal Grammars." In Proceedings of the 4th International Conference on Software Engineering. IEEE, New York, NY, USA, 1979, pp.425-432.
- [Berners-Lee et al. 96] Berners-Lee, T., Fielding, R., Frystyk, H. "Informational RFC 1945 - Hypertext Transfer Protocol -- HTTP/1.0," Request for Comments, Internet Engineering Task Force, May 1996.
- [Celentano+ 80] Celentano, A., Crespi-Reghizzi, S., Vigna, P. D., Ghezzi, C., Granata, G., and Savoretti, F. "Compiler Testing Using a Sentence Generator." In Software: Practice & Experience. 10(11), 1980, pp. 897-918.
- [Chang et al. 95] Chang, J., Richardson, D. J., and Sankar, S. "Automated Test Selection from ADL Specifications." In the First California Software Symposium (CSS'95), March 1995.
- [Dean et al. 97] Dean, D., Felten, E. W., Wallach, D. S. and Belfanz, D. "Java Security: Web Browsers and Beyond." In Internet Beseiged: Countering Cyberspace Scofflaws, D. E. Denning and P. J. Denning, eds. ACM Press, October 1997.
- [Dershowitz+ 90] Dershowitz, N. and Jouannaud, J. Rewrite Systems. In Handbook of Theoretical Computer Science. Volume B: Formal Methods and Semantics. Van Leeuwen, ed. Amsterdam 1990.
- [Dershowitz 93] Dershowitz, N. A Taste of Rewrite Systems. In Lecture Notes in Computer Science 693, 199-228 (1993).
- [Drossopoulou+ 97] Drossopoulou, S., Eisenbach, S. "Java Is Typesafe - Probably." In 11th European Conference on Object Oriented Programming, June 1997.
- [Drossopoulou+ 98] Drossopoulou, S., Eisenbach, S. "Towards an Operational Semantics and Proof of Type Soundness for Java." March 1998, to be published. <http://www-dse.doc.ic.ac.uk/projects/slurp/pubs/chapter.ps>.

- [Drossopoulou+ 99] Drossopoulou, S., Eisenbach, S. and Khurshid, S. "Is the Java Typesystem Sound ?" In *Theory and Practice of Object Systems*, Volume 5(1), p. 3-24, 1999.
- [Freund&Mitchell 98] Freund, S. N. and Mitchell, J. C. "A type system for object initialization in the Java Bytecode Language." In *ACM Conference on Object-Oriented Programming: Systems, Languages and Applications*, 1998.
- [Freund&Mitchell 99] Freund, S. N. and Mitchell, J. C. "A Type System for Java Bytecode Subroutines and Exceptions." To be published.
- [Gargantini+ 99] Gargantini, A. and Heitmeyer, C. "Using Model Checking to Generate Tests from Requirements Specifications." In *Proceedings of the Joint 7th Eur. Software Engineering Conference and 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Toulouse, France, September 1999.
- [Griswold] Griswold, D. "The Java HotSpot Virtual Machine Architecture." <http://www.javasoft.com/products/hotspot/whitepaper.html>
- [Inferno] Lucent Technologies. Inferno. <http://inferno.bell-labs.com/inferno/>
- [Lindholm&Yellin 99] Lindholm, T. and Yellin, F. *The Java Virtual Machine Specification*, 2nd Ed. Addison-Wesley, 1999.
- [Mandrioli et al. 95] Mandrioli, D., Morasca, S., and Morzenti, A. "Generating Test Cases for Real-Time Systems from Logic Specifications." *ACM Transactions on Computer Systems*. Vol.13, no. 4, November 1995, pp. 365-398.
- [McGraw & Felten 96] McGraw, G. and Felten, E. W. *Java Security: Hostile Applets, Holes and Antidotes*. John Wiley and Sons, New York, 1996.
- [McKeeman 98] McKeeman, W. M. "Differential Testing for Software." *Digital Technical Journal* Vol 10, No 1, December 1998.
- [Meyer & Downing 97] Meyer, J. and Downing, T. *Java Virtual Machine*. O'Reilly, March 1997.
- [Muller et al. 97] Muller, G., Moura, B., Bellard, F., Consel, C. "Harissa: A Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code." In *Proceedings of the Third Conference on Object Oriented Technologies*, 1997.
- [Necula 97] Necula, G. "Proof Carrying Code." In *Principles of Programming Languages*. Paris, France, January 1997.
- [Necula & Lee 98] Necula, G. and Lee, P. "The Design and Implementation of a Certifying Compiler." In *Programming Languages, Design and Implementation*. Montreal, Canada, June 1998.
- [Oracle] Oracle Corporation. Oracle Application Server Release 4.0 Documentation. <http://technet.oracle.com/doc/was.htm>
- [Proebsting et al. 97] Proebsting, T. A., Townsend, G., Bridges, P., Hartman, J.H., Newsham, T. and Watterson S. A. "Toba: Java for Applications: A Way Ahead of Time (WAT) Compiler." In *Proceedings of the Third Conference on Object Oriented Technologies*, 1997.
- [Prusinkiewicz et al. 88] Prusinkiewicz, P., Lindenmayer, A., Hanan, J. "Developmental Models of Herbaceous Plants for Computer Imagery Purposes." In *Computer Graphics*, 22 (4), August 1988.
- [Sirer et al. 98] Sirer, E. G., Grimm, R., Bershad, B. N., Gregory, A. J. and McDirmid, S. "Distributed Virtual Machines: A System Architecture for Network Computing." *European SIGOPS*, September 1998.
- [Stata&Abadi 98] Stata, R. and Abadi, M. "A type system for Java bytecode subroutines." In *Proceedings of the 25th ACM Principles of Programming Languages*. San Diego, California, January 1998, p. 149--160.
- [SunJWS] Sun. Java Web Server. <http://www.sun.com/software/jwebserver/index.html>
- [Syme 97] Syme, D. "Proving Java Type-Soundness." *University of Cambridge Computer Laboratory Technical Report 427*, June 1997.
- [Weyuker et al. 94] Weyuker, E., Goradia, T., and Singh, A. "Automatically Generating Test Data From a Boolean Specification." In *IEEE Transactions on Software Engineering*, Vol. 20, no. 5 (May 1994), 353-363.

Jargons for Domain Engineering

Lloyd H. Nakatani, Mark A. Ardis,
Robert G. Olsen, Paul M. Pontrelli
Lucent Technologies USA

Abstract

In the Family-oriented Abstraction, Specification and Translation (FAST) domain engineering process for software production, a member of a software product family is automatically generated from a model expressed in a DSL. In practice, the time and skill needed to make the DSLs proved to be bottlenecks. FAST now relies on jargons, a kind of easy-to-make DSL that domain engineers who are not language experts can quickly make themselves. We report our experiences with jargons in the FAST process, and describe the benefits they provide above and beyond conventional DSLs for software production and other purposes.

1. Introduction

We report here our experience with jargons [Nakatani97] for software engineering. Jargons are DSLs that are unusually easy to make. We use jargons within the framework of the Family-oriented Abstraction, Specification and Translation (FAST) domain engineering process [Parnas76] [Cuka98] to automate software production. Previous attempts with FAST had foundered when the DSLs needed for domain modeling took too long to make. Because jargons can be made quickly and easily, they seemed a good alternative to conventional DSLs.

Preliminary experiments made us optimistic that jargons would work for FAST. We made jargon equivalents of two existing DSLs that had each taken over a year to make, even using language implementation tools such as yacc [Johnson75]. The results were dramatic. Each jargon took less than a week to make. The catch was that the jargon maker in the experiments was the inventor of jargons. Further work was needed to see if jargons could be made as quickly by software developers who were domain experts but were not experts in language design and implementation, and had no prior experience with jargons. We believe that jargons are most likely to succeed when developers can make their own.

Our effort was focused on getting answers to the following questions:

- Can jargons handle the complexity of real world domains?
- Can domain experts make their own jargons?
- Do jargons avoid the major pitfalls of DSLs?

Of course, we were open to unexpected discoveries, good and bad.

The FAST process is described in section 2, jargons in section 3, the software domain in section 4, benefits of jargons in section 5, potential pitfalls of DSLs in section 6, related work in section 7, and conclude with lessons learned in section 8.

2. FAST Process for Domain Engineering

Many software products constitute a family consisting of many variations of essentially the same thing. The variations may be successive generations of a product, or different versions of a product for different platforms, customers, or market segments. A familiar example of a hardware family is a car that comes in stripped-down or luxury versions and in a choice of two- or four-door models. We believe that software families are ubiquitous. However, good examples are hard to come by, because software is usually not described in those terms. An example from telecommunications is the 5ESS(RM) electronic switch software [Martersteck85] that comes in different versions customized for different customers and hardware configurations.

FAST exploits the properties of software families to make the production of family members more efficient. The FAST process is split into two phases: domain engineering, and application engineering. (See Figure 1.)

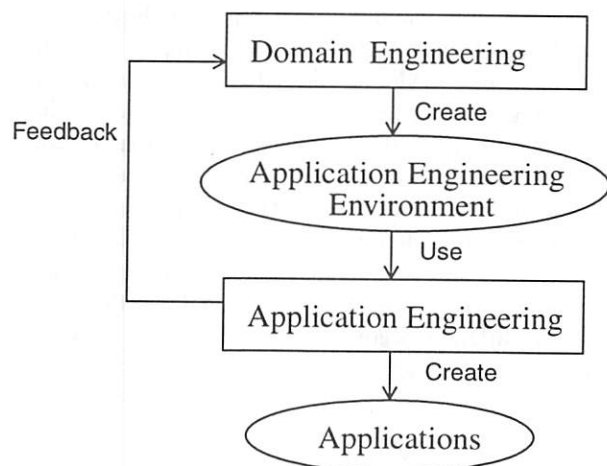


Figure 1. FAST process of domain engineering

From the description below, one might get the impression that FAST is a serial process, but in fact it is an iterative process in practice with much feedback between the phases.

In the *domain engineering* phase, family members are analyzed to discover their commonalities—what is the same about every member—and their variabilities—what differentiates one member from another. A software architecture reflecting the commonalities and variabilities is then designed.

The parts constituting the commonalities are expected to remain constant over family members. There is little need to optimize their production, since they will only be created once. On the other hand, integration of these parts with other software is a concern. Some FAST projects elect to automate the generation of commonalities code in order to simplify later integration.

The variabilities are addressed in the domain engineering phase by making one or more jargons to model family members solely in terms of their variabilities.

Translators are written to translate models into variabilities code. The use of high-level modeling languages, of which jargons are instances, distinguishes FAST from other domain engineering processes.

In the *application engineering* phase, a model of a particular family member is written in terms of one or more jargons, and then translated automatically into variabilities code. Finally, the commonalities code and variabilities code are integrated to produce the product. The integration process, for instance, may be building a load module incorporating the compiled commonalities and variabilities codes.

Two aspects of FAST have important ramifications for jargons. First, partitioning a product into its commonalities and variabilities is key to making automation feasible. The commonalities usually represent the core of the product and encapsulate most of its complexities. The commonalities code and variabilities code are separated, allowing for independent development. The variabilities part is usually much smaller and simpler than the commonalities, so automating its production is easier than automating the production of the entire product. Second, the variabilities often represent distinctly different features of the product, and may not lend themselves to modeling with a single jargon. When this is the case, several different jargons may be necessary to model the distinct variabilities.

3. Jargons: Domain Engineered DSLs

In essence, jargons are domain engineered DSLs. The expected benefit is a production environment called InfoWiz for making jargons efficiently. With InfoWiz, we can make a jargon in a matter of days or weeks instead of months or years for conventional DSLs. Some unexpected benefits are: 1) jargon making simplified to a Do-It-Yourself (DIY) activity; 2) composable jargons (i.e., different jargons are compatible and can be used in combination); and 3) multipurpose models (i.e., multiple products can be produced from a single model written in a jargon). So compared to conventional DSLs, jargons provide more benefits at less cost.

Jargons comprise a family of custom-made DSLs with the following commonalities:

- Abstract syntax: Every expression of every jargon has the same abstract syntax.
- Generic interpreter: All jargons are processable with the same generic interpreter specialized at runtime with the semantics of the pertinent jargons.

A jargon is distinguished from another by the following variabilities:

- Concrete syntax: A set of concrete expressions
- Semantics: A set of actions corresponding to the expressions that define a semantics of the jargon

InfoWiz, the application engineering environment for making jargons, consists of the following components:

- WizTalk abstract syntax
- InfoWiz generic interpreter
- Fit programming language for defining actions
- API functions for interfacing actions to the InfoWiz interpreter

This section describes these components to show what jargons are like, and how they are designed, implemented and used.

3.1. WizTalk Abstract Syntax

The WizTalk abstract syntax prescribed for every expression of every jargon is

```
; term(note-1 | ... | note-N) [memo]
```

The `;` metacharacter is a *marker* that distinguishes jargon expressions from *plaintext*, which is any text that is not a jargon expression. The marker makes possible *markup jargons* with expressions embeddable in plaintext; for example

```
The ;cb[;] metacharacter is a  
;i[marker]
```

This is the beginning of this paragraph in the markup jargon used to format a draft of this paper.

The *term* is the name of the expression. The jargon

designer is free to choose concrete terms that best reflect the natural terminology of the domain. If care is taken to make the concrete terms unique across all jargons that might collaborate, the jargons are composable without conflict. The term is case-sensitive, and `.` and `_` can be used interchangeably as separators between words of a multi-word term.

The *memo* is the information that is the focus of the expression. WizTalk allows three syntactic variants of a memo. When the memo fits on a single line, the preferred variant is an *inline memo*:

```
;author[Mark Twain]
```

The `[` and `]` metacharacters are *memo delimiters*. When the memo consists of multiple lines, one variant is a *block memo*:

```
;address[
123 Main Street
Anyville, NJ 01234
]
```

where the memo and its closing `]` delimiter are aligned with the margin established by the indentation of the line containing the expression. An alternative variant for a multi-line memo is an *inset memo*:

```
;address
    123 Main Street
    Anyville, NJ 01234
```

where the memo is tab-indented one level deeper than the indentation of the line beginning the expression. Indentation is thus syntactically significant. Inset memos make the hierarchical structure of complex models easy to see.

The *notes* are either attributes of the memo, or parameters to control the processing of the expression. For example, in

```
;state(END)
;exit
```

the note is the name of the state of a finite state machine. The `(` and `)` metacharacters are *note delimiters*. An expression may have neither note nor memo; the `;exit` expression above is an example.

The WizTalk abstract syntax has proven versatile enough for markup jargons to model the format of documents, data jargons to model hierarchically structured information, and programming jargons to model algorithms.

3.2. InfoWiz Generic Interpreter

The InfoWiz generic interpreter can process any jargon when customized with a semantics of the jargon. A semantics of a jargon is defined by a set of *actions*, one for each expression of the jargon. An action, which is a function written in the Fit programming language, specifies how the information associated with an expression should be processed. A file containing a set of action definitions is called a *wizer*. The InfoWiz interpreter, which is also written in Fit, customizes itself by incrementally loading one or more wizers, and automatically integrating the actions they contain.

Once customized, the InfoWiz interpreter processes a model written in the jargon by parsing the model; traversing the parse tree in top-down, left-right, depth-first order; executing the action corresponding to the expression at each node of the parse tree in traversal order, and appending the expression's product to an output buffer. Plaintext is a degenerate expression whose product is the verbatim text. A scalar product for a parent expression is produced by concatenating the products accumulated in the output buffer for the child expressions in the memo of the parent expression.

Actions are described in more detail later, but a tiny example is presented here to illustrate how the InfoWiz interpreter is used. This jargon document in file `greet.doc`

```
;greet[InfoWiz]
```

consists entirely of the `;greet` expression. This action in the wizer file `hello.w`

```
A_greet
    WizOut "Hello, " GetWizMemo
```

is a Fit function that defines a semantics for the `;greet` expression. The action name consists of the `A_` prefix followed by the expression's term. The `GetWizMemo` API function processes the memo of the expression, and returns the product, which for this example is the plaintext `InfoWiz`. The `WizOut` API function splices together its arguments and appends the result to the output buffer. The `wiz` command

```
$ wiz -w hello.w greet.doc
Hello, InfoWiz
```

runs the InfoWiz interpreter that integrates the actions in the `hello.w` wizer, processes the document in `greet.doc`, and writes the product to the standard output. The `-w` command option identifies `hello.w` as the wizer.

3.3. Fit Programming Language and API Function Library

The Fit programming language makes it possible to write actions quickly with the support of the API function library. Fit is a high-level, general-purpose, interpreted programming language developed at Lucent and AT&T. It is beyond the scope of this paper to explain Fit. Suffice it to say that Fit has excellent facilities for text processing, supports multiple programming paradigms including object-oriented, and has a source code debugger integrated into its interpreter.

The API functions, which are written in Fit, enable actions to interface with the InfoWiz interpreter. The API library consists of less than 50 functions. Of these, about a dozen are frequently used.

3.4. Multipurpose Models

Because actions are easy to write and their integration into the InfoWiz interpreter is automated, it has proved practical for a jargon to have multiple semantics. The consequence has been multipurpose models capable of producing many different products. If yet another product is needed from a model, it can be produced simply by writing a new wizer and processing the model with the wizer.

We use this greeting model from before

```
;greet[InfoWiz]
```

to illustrate how multipurpose models work in practice. We define another semantics for the `;greet` expression with this action:

```
Macro HELLO ~[
main()
{
printf("%s", \
    "Hello, <greetee>\n" );
}
]~

A_greet
WizOut Change "<greetee>"
(GetWizMemo) HELLO
```

When the greeting model is processed with this new semantics, the result is a C program that prints Hello, InfoWiz:

```
$wiz -w Chello.w greet.doc
main()
{
    printf( "%s", "Hello,
InfoWiz\n" );
}
```

Multipurpose models ensure that all the products generated from a model are consistent with each other. For example, if the greeting model is changed to

```
;greet[World]
```

this change is reflected in all the products obtained from the model.

3.5. Composability of Jargons

Because all jargons have a common abstract syntax, multiple jargons can be processed by the InfoWiz interpreter simultaneously customized with the semantics of the jargons. In effect, this makes all jargons composable and able to collaborate with each other to solve problems beyond their individual reach.

To illustrate jargon composition in practice, we use this finite state machine model of an interactive dialogue:

```
;state(START)
;next.state[INPUT]
;state(INPUT)
;=x[;input[Type name: ]]
;if( ;.x == q )
;next.state[GOOD BYE]
;else
;next.state[HELLO]
;state(HELLO)
;output
;frame
;star[Hello, ;.x]
;next.state[INPUT]
;state(GOOD BYE)
;output[Good bye]
;next.state[END]
;state(END)
;exit
```

This model is composed of four jargons: the *FSM*, *Base*, *Flow*, and *Banner* jargons: The *FSM* jargon for modeling finite state machines consists of the `;state`, `;next.state`, and `;exit` expressions. The *Base* jargon, which comes standard with the InfoWiz interpreter, includes the `;=x[InfoWiz]` expression to set variable `x` to InfoWiz, and the `;.x` expression to get its value, among others of similar generic nature that are likely to be useful for many domains. The *Flow* jargon for modeling algorithms includes flow control expressions such as `;if` and `;else`, the `;input` and `;output` expressions, and relational predicates using infix notation such as `;.x == q` to test whether variable `x` has value `q`. The *Banner* jargon for modeling "banners" such as

```
=====
| *** Hello, InfoWiz *** |
=====
```

includes the `;frame` expression to put a frame around a message, and the `;star` expression to bracket a message with `***`.

Because the model is composed of multiple jargons, multiple wizers are needed for its execution:

```
$ wiz -w flow -w fsm.w \
      -w banner.w hello.doc
Type name: InfoWiz
=====
| *** Hello, InfoWiz *** |
=====
Type name: q
Good bye
$
```

The results of the execution is an interactive dialogue shown above.

To illustrate how easy wizers are to write, shown below are the wizers for two of the jargons. This is the wizer for the *FSM* jargon

```
A_state label
  Local fsm
  Set fsm[label] GetWizTree

A_next_state
  Local fsm state
  Set state fsm[GetWizMemo]

A_exit
  Local state
  Set state nil

WizAfter out
  Local fsm state
  Set state fsm["START"]
  While state
    WizMemo state
```

This is the wizer for the *Banner* jargon

```
A_star
  WizOut "**** " \
        (GetWizMemo) " ****"

A_frame
  Set message Ssplice "| " \
        (GetWizMemo) " |"
  Set edge<Thru 1 $message> \
    "="
  WizOut |"\n"| edge message \
    edge
```

For the lack of space, we provide no further explanation of the wizers.

4. Configuration Control in the 5ESS

The 5ESS is a highly reliable telecommunications switch. The reliability stems in part from redundant hardware and automatic reconfiguration software that removes faulty hardware units from service and replaces them with their spares. The Configuration Control domain, which includes this reconfiguration software, was re-engineered with the FAST process to speed up software production. Our report is based mostly on our experience with jargons in the Configuration Control domain.

An analysis of the Configuration Control domain resulted in a clean separation between the configuration and the algorithms that change the configuration. The configuration is a variability of the domain since different switches are composed of different hardware units, and the units can be in different conditions (e.g., *ready* or *working*). The reconfiguration algorithms are operations that change the configuration. They are the commonalities of the domain, since the algorithms are the same for all units.

Per the FAST process, a Configuration jargon was made to model a configuration. A configuration was representable as a graph with nodes corresponding to the hardware units, and edges representing the relationship between the units. A wizer was written to translate a configuration model into declarations of populated C data structures that represented the hardware configuration of a switch in a form suitable for the reconfiguration algorithms. To facilitate the translation, a Cstruct jargon was made to model the C data structures. The translator was modeled by the composition of both models: the configuration model for the source, and the C data structure model for the target.

In order to separate the common parts from the variable parts of the reconfiguration algorithms, a Controller jargon was made. In addition to separating commonalities from variabilities, this jargon made it possible for other domain experts to check the accuracy of the algorithms. Each algorithm was modeled in the jargon, translated into a finite state machine, and from there to C code. When a bug was found in the algorithm code, its model was fixed, and new code regenerated. From jargon models totalling about 500 lines, over 50,000 lines of C code were generated.

In the final integration process, the codes for the configuration data structures and the reconfiguration algorithms were compiled together along with handwritten code that handled the interface between the two, and also between the Configuration Control domain and other domains.

5. Benefits of Jargons in Domain Engineering

Jargons provide many benefits to the FAST domain engineering process. In this section we describe attributes of jargons that we believe were key to their success in our environment.

5.1. Domain Decomposition and Modeling

Jargons fostered a decomposition (divide-and-conquer) strategy for coping with the complexity of a domain. For the Configuration Control domain, the first decomposition divided the domain into its commonalities (reconfiguration algorithms) and its variabilities (configuration) per the FAST process. The configuration subdomain further divided into the configuration itself and the C data structures into which the configuration had to be translated. Three jargons were made in accordance with this decomposition: the Controller jargon for modeling the reconfiguration algorithms; the Configuration jargon for modeling the configuration; and the Cstruct jargon for modeling C data structures. The Configuration jargon was a natural by-product of the domain analysis, and bears a strong resemblance to a section of the analysis document. Very little effort was required to design this language. Similarly, the Cstruct language was modeled on existing data structure facilities of the C language, so it was easy to design.

Two other domains—the Call Billing and Measurements domains—re-engineered with the FAST process also decomposed naturally into multiple subdomains. The Call Billing domain produces software that generates accounting records for billing calls. This domain decomposed into four subdomains: 1) algorithms that determine the kind of record to be generated for a call, 2) the format and contents of the fields comprising a record, 3) the concrete values of abstract variables that could populate a field, and 4) the attributes of the abstract variables. A separate jargon was made for modeling the artifacts of each subdomain. The Measurements domain produces the software that reports on the performance of equipment in a telephone office. The domain has already decomposed into four subdomains, with more expected. Three of the subdomains have their own jargons, and the fourth was further decomposed into three subdomains with their own jargons. So altogether, there are six tiny jargons for modeling the subdomains of the Measurements domain.

Decomposition works only if the solutions for the parts can be composed into a solution for the whole. For our approach, this means that the subdomain models expressed in different jargons must be composable into larger models for the entire domain. Model composition

was the rule in every domain. In the Configuration Control domain, the configuration and reconfiguration algorithm models were composed to build a simulator to test the algorithms on real data. In the Call Billing domain, all the models had to be composed in order to generate the records. And in the Measurements domain, models of the measurement, report content, and report structure had to be composed to generate the reports.

5.2. Do-It-Yourself Jargons

With InfoWiz, a jargon is so easy to make that domain experts can make their own. Such Do-It-Yourself (DIY) jargons can be expected to provide the following benefits:

- Better jargons
- Lower risk
- Easier maintenance

A DIY jargon made by a domain expert is likely to turn out better than one made by a language expert who is not a domain expert for several reasons. First and foremost, the domain expert knows the domain: the “natural jargon” spoken among experts, the subdomains of the domain, what has to be modeled in each subdomain, how the models should be composed, what products have to be generated, the assumptions needed to translate models into products, the legacy languages of the products, and how the generated products integrate with other code, such as the commonalities code. Second, the domain expert and the end-users of the jargon usually belong to the same organization. The easy communication between the jargon maker and jargon users makes for an ideal rapid prototyping environment. Jargons lend themselves to rapid prototyping because of the ease with which they can be made and modified. The domain expert can take good advantage of the environment and rapid prototyping to make a “user friendly” jargon.

A DIY jargon is less risky. When the organization makes and supports its own jargons, the risk that comes from depending on outside experts is eliminated. Bugs can be fixed immediately, and new features added as needed, no matter how minor. In event of a crisis years down the road when the outside expert is no longer available, the organization will have the wherewithal to cope.

Compared to general-purpose programming languages, jargons can be expected to evolve rapidly, which puts a premium on easy evolution. A jargon is custom-made for modeling a particular domain, so as the domain evolves to meet changing needs, its jargon must evolve to keep up. It follows that jargons must be easy to maintain if they are not to become quickly obsolete [VanDuersen97] [Spinellis97].

Jargon making is indeed simple enough to be a DIY activity. In the Configuration Control domain, the jargons were created by a team of three domain experts and one InfoWiz expert. The domain experts produced and maintain all of the jargons. Three completely different versions of one jargon were prototyped in a month. In the Call Billing domain, the jargons and translators were the result of collaboration between an InfoWiz expert and a domain expert. However, another domain expert added major new capabilities to one of the jargons. In the Measurements domain, all of the jargons were made by domain experts with an InfoWiz expert serving as a reviewer only. None of the domain experts had any previous experience in making languages.

DIY jargons are possible because most of the hard work is already done. Skill in the art and science of syntax design and grammar specification is unnecessary because WizTalk prescribes a ready-made abstract syntax for all jargons. Skill in the art and science of building a parser and interpreter is unnecessary because the ready-made, generic InfoWiz interpreter works for all jargons. Defining the semantics of a jargon is reduced to writing a set of actions in a high-level, interpreted programming language. No work is necessary to integrate the actions into the interpreter because the integration is automatic. In the Configuration Control domain, the domain experts mastered jargon technology within a few days.

5.3. Raising Software Quality

Jargons go beyond conventional DSLs to improve the quality of software products. Jargons share with conventional DSLs the benefits of specialization, high-level abstractions, and automatic generation to eliminate accidental errors. Jargons go beyond these conventional techniques to improve software quality by the following means:

- Simplification through decomposition of a complex domain into subdomains
- Generation of related subproducts from a single source to ensure their consistency
- Transformation of models into more readable forms for review
- Automatic model checking with user-defined types

The decomposition of a complex domain into simpler subdomains, and modeling each in its own jargon, improves quality indirectly. The simpler the domain, the simpler their jargons, and the simpler the jargons, the more likely are their designs and translators to be correct. Moreover, simple models expressed in simple jargons are shorter and easier to write, review, and debug. In all of the domains we've worked on, decomposition

has resulted in simplifications that most likely improved the quality of the final product. However, without metrics of complexity or simplicity, it's hard to attribute any improvement in quality to the simplification that comes from decomposition.

One measure of the quality of a product is the consistency between its subproducts. Consistency is ensured if all the subproducts are produced from a single multipurpose model. Multipurpose models were used to good effect in the reconfiguration subdomain of the Configuration Control domain. Two human-friendly representations of each reconfiguration algorithm were generated from its model: a pictorial representation as a flow chart, and a stylized English representation. Reviewers preferred the pictorial and natural language representations because they were easier to comprehend. The fact that the review representations and the software were generated from the same model ensured that the product approved by the reviewers was what was actually produced. In addition, the graphical tool used by application engineers was enhanced with a simulator that showed the behavior of the reconfiguration algorithms. The original models of the algorithms were translated by a wizer into an internal finite state machine model in the language of the graphical tool. This guaranteed that the behavior of the simulator would remain consistent with the generated code.

A specification of a jargon can be written in a ready-made Checker jargon that comes standard with InfoWiz. Such a specification expressed is the analog of a Document Type Description in SGML [Goldfarb90]. A specification is translated into a wizer that defines a self-checking semantics for the jargon. When a model is processed with this semantics, each expression checks itself against its specification. No specification was written for the jargons of the Configuration Control domain because the Checker jargon was not available at the time. In retrospect, specifications of the Configuration Control jargons would have been of limited use. Configuration models are now being generated by an interactive tool with a graphical user interface (GUI) that prevents the kinds of mistakes that would be caught by a model checker. The tool makes a checker of little value. Algorithm models require analysis and checks of its dynamic behavior. A specification would only do a static analysis and checks that would be of limited value in detecting bugs in the algorithms. The value of a specification goes beyond checking because it also serves as documentation for users of a jargon.

6. Avoiding DSL Pitfalls with Jargons

Some pitfalls lie in the path leading to the widespread use of DSLs. Unless these pitfalls are successfully avoided, we feel that the use of DSLs will prove counter-productive in the long run. Jargons avoid these pitfalls.

6.1. Pitfall: Balkanized Domains

Conventional DSLs are not composable. As a consequence, different DSLs cannot work together just as Lisp, C, and Java cannot. DSLs make problems in their respective domains easy to solve, but their impossibility will make problems involving multiple domains even harder to solve by preventing collaboration between domains that are each part of the solution. It is critical to avoid this pitfall if DSLs are to be a step forward rather than backward.

Composability distinguishes jargons from conventional DSLs. Jargons are composable because their common syntax makes all jargons processable with a common interpreter, and because the interpreter is customizable with the semantics of multiple jargons simultaneously. With composable jargons, we can prevent the Balkanization of domains.

6.2. Pitfall: Cost

The cost of DSLs is a potential pitfall. A large company that embraces DSLs may end up with hundreds of DSLs instead of just a few general-purpose languages. General-purpose languages are supported by their vendors, but DSLs will most likely be supported by their users. As DSLs proliferate, their aggregate support costs can get out of hand.

Compared to a conventional DSL, a jargon is inexpensive to make, because much of the work is already done. The syntax is already designed, and the interpreter is already written. Making a jargon is reduced to designing its concrete expressions, and writing the actions for the expressions in a high-level programming language with excellent debugging facilities.

Compared to a conventional DSL, a jargon is also inexpensive to maintain. Any jargon can be extended without changing its abstract syntax, which means that the interpreter doesn't have to be changed to accommodate the extensions. Only one interpreter and one API library need be maintained for all jargons. The commonalities among jargons also help to reduce other support costs, such as for training and documentation.

The composability of jargons has ramifications for their cost and timeliness. If a problem involving multiple

domains must be solved, and jargons already exist for the domains, then it may be possible to solve the problem by composing the existing jargons, thereby eliminating entirely the cost and time of making yet more jargons.

7. Related Work

Jargons are instances of little languages [Bentley86]. As such, they share the attributes of simplicity and ease-of-use that are the hallmarks of domain-specific specialization. Unlike making a jargon, making a conventional little language entails all the steps of making a "big" language: syntax design, grammar specification, parsing, and execution. These differences loom large in practice. We have found that developers who cannot make a little language can easily make a jargon in a day or two. And once made, jargons provide the advantages of composability and multipurpose models that little languages do not.

XML is closest in spirit to jargons. Languages based on XML could be used for domain engineering, but their implementations are biased toward document markup languages with their style sheets and style sheet languages for defining their semantics. This bias is not well matched to the needs of domain engineering where models are typically not documentation. Moreover, the back-end of interpreters for XML-based languages are harder to construct than for jargons. That said, the similarity between jargons and XML demands that we distinguish the InfoWiz concept from its implementation. The InfoWiz concept is founded on the following notions:

- A constant abstract syntax for all jargons
- A multiplicity of easily defined computational semantics for a given jargon
- A generic interpreter that is easily customizable with the computational semantics of multiple jargons simultaneously

The InfoWiz concept could be implemented with XML instead of WizTalk for the jargon syntax, and Perl or Java or Python instead of Fit for the host programming language. It would be good to have different implementations of the InfoWiz concept to suit different needs and tastes.

Spinellis and Guruprasad [Spinellis97] describe by way of numerous examples how software engineering is facilitated with DSLs. Despite their success, we feel that the use of conventional DSLs of the sort they advocate is ill-advised. A conventional DSL is hard to make, costly to maintain, and becomes yet another obstacle to teamwork among domains.

Faith et al. [Faith97] describe the Khepera system for making DSLs. Khepera gives the maker of a DSL the freedom to design its syntax. But our developers, who were not language experts, considered syntax design, grammar specification, and tools such as `yacc` as obstacles, not opportunities. Moreover, DSLs with different syntax are guaranteed to be impossible. Also, the transformation approach of Khepera requires more understanding of abstract syntax trees, tree traversal and manipulation, and the inner workings of an interpreter than our developers had.

A domain-specific embedded language [Hudak96] is another approach to speed up software production. But this approach works only for languages that support rich abstraction mechanisms that make it possible to extend the base language with domain-specific extensions. Unfortunately, for reasons of legacy, our applications must be written in C. Since C does not support the abstractions needed to realize domain-specific embedded languages, we are unable to use this approach. We find such legacy constraints more the rule than the exception.

8. Lessons Learned

Jargons are up to the challenges of real-world domains. The following divide-and-conquer strategy fostered by easy-to-make jargons worked for each domain:

- Decompose the domain into subdomains
- Make a jargon for each subdomain
- Model each subdomain in its jargon
- Compose the subdomain models to model the entire domain

Although every jargon has the same abstract syntax, we found the syntax versatile enough to meet the expressive demands of their subdomains.

Domain experts can make their own jargons. In fact, we believe that domain experts make the best jargons because they have the necessary knowledge, and because they are in an environment that is conducive to rapid prototyping.

Jargons improve the quality of software. The decomposition of domains leads to higher quality software because simple models expressed in simple jargons are easier to get right. And when the simple models are composed, the resulting whole is more likely to be correct because it is built of proven components. Multipurpose models can generate multiple subproducts from a single source to ensure their consistency. Models can be translated into more comprehensible representations for review. This makes flaws in the models easier to catch. And because both the review representation and final product are generated from a single source, what is

approved is what will be produced.

Jargons avoid the key pitfalls of conventional DSLs. As jargons proliferate, their aggregate cost is minimized because they are easy to make, and only one InfoWiz interpreter and API function library need be maintained for all jargons. The composability of jargons prevents the Balkanization of domains, and leverages the power of specialization through teamwork.

9. Acknowledgments

We thank the many domain experts who contributed to the success of the Configuration Control domain engineering project. In particular, we thank Paul Iverson and Andy Kranenborg for their help during the domain analysis phase. We thank Mehry Moukhtar, Dan Johnson, Michelle Homer, Tom Denton, Lynn Pautler, and Carl Amport for their support and encouragement. For contributions to the Call Billing domain engineering project we thank Mike Moy, Christine Fischer, Lyn Cole, and Steve Powell. For contributions to the Measurements domain engineering project we thank Roman Biesiada, Przemyslaw Marciniak, Hanna Weber, Yanti Miao, and Diane Kruto. We thank David Cuka for his pioneering work with InfoWiz and FAST on several projects. Finally, we thank David Weiss for creating and supporting the FAST process at Bell Labs.

10. Availability

InfoWiz and the jargons described here are the proprietary property of Lucent Technologies. The programming language Fit is in the public domain. It is available by sending email to lwr@research.att.com. Contact PaceLine Technologies (www.paceline-tech.com) about obtaining InfoWiz.

11. References

- [Bentley86] Bentley, J. Little Languages, *Communications of the ACM* **29** (8), August 1986, 711-721.
- [Cuka98] Cuka, D.A. and Weiss, D.M. Engineering Domains: Executable Commands as an Example, *Proceedings 5th International Conference on Software Reuse*, Victoria, Canada, June 2-5, 1998, 26-34.
- [Faith97] Faith, E.R., Nyland, L.S. and Prins, J.F. Khepera: A System for Rapid Implementation of Domain-Specific Languages, *Proceedings of the Conference on Domain-Specific Languages*, Santa Barbara, CA, October 15-17, 1997, 243-255.
- [Goldfarb90] Goldfarb, Charles F. *The SGML Handbook*, Clarendon Press, Oxford, England, 1990.
- [Hudak96] Hudak, P. Building Domain-Specific Embedded Languages, *ACM Computing Surveys* **28** (4), December 1996.
- [Johnson75] Johnson, S.C. Yacc --- Yet Another Compiler-Compiler, *Computer Science Technical Report 32*, Bell Laboratories, July 1975.
- [Martersteck85] Martersteck, K.E. and Spencer, A.E. Introduction to the 5ESS(RM) Switching System, *AT&T Technical Journal* **64** (6), July-August 1985, 1305-1314.
- [Nakatani97] Nakatani, L.H. and Jones, M.A. Jargons and Infocentrism, *Proceedings of DSL '97 (First ACM SIGPLAN Workshop on Domain-Specific Languages)* Paris, January 18, 1997, 59-74. Published as University of Illinois Computer Science Report, <http://www-sal.cs.uiuc.edu/~kamin/dsl>.
- [Parnas76] Parnas, D. L. On the Design and Development of Program Families, *IEEE Transactions on Software Engineering* **2**, 1976, 1-9.
- [Spinellis97] Spinellis, D. and Guruprasad, V. Lightweight Languages as Software Engineering Tools, *Proceedings of the Conference on Domain-Specific Languages*, Santa Barbara, CA, October 15-17, 1997, 67-76.
- [VanDuersen97] Van Duersen, A. and Klint, P. Little Language, Little Maintenance? *Proceedings of DSL '97 (First ACM SIGPLAN Workshop on Domain-Specific Languages)*, Paris, January 18, 1997, 109-127. Published as University of Illinois Computer Science Report, <http://www-sal.cs.uiuc.edu/~kamin/dsl>.
- [XML] Anonymous. The XML Information Site, <http://www.xmlinfo.com>

Slicing Spreadsheets: An Integrated Methodology for Spreadsheet Testing and Debugging

James Reichwein, Gregg Rothermel, Margaret Burnett
Department of Computer Science
Oregon State University
Corvallis, OR 97331
{reichwja,grother,burnett}@cs.orst.edu

Abstract

Spreadsheet languages, which include commercial spreadsheets and various research systems, have proven to be flexible tools in many domain specific settings. Research shows, however, that spreadsheets often contain faults. We would like to provide at least some of the benefits of formal testing and debugging methodologies to spreadsheet developers. This paper presents an integrated testing and debugging methodology for spreadsheets. To accommodate the modelless and incremental development, testing and debugging activities that occur during spreadsheet creation, our methodology is tightly integrated into the spreadsheet environment. To accommodate the users of spreadsheet languages, we provide an interface to our methodology that does not require an understanding of testing and debugging theory, and that takes advantage of the immediate visual feedback that is characteristic of the spreadsheet paradigm.

1 Introduction

Spreadsheet languages, which include commercial spreadsheet systems as a subclass, have proven useful in many domain specific settings, including business management, accounting, and numerical analysis. The spreadsheet paradigm is also a subject of ongoing research in many domain specific areas. For example, there is research into using spreadsheet languages for matrix manipulation problems [33], for providing steerable simulation environments for scientists [7], for high-quality visualizations of complex data [9], and for specifying full-featured GUIs [21].

Despite the end-user appeal of spreadsheet languages and the perceived simplicity of the spreadsheet paradigm, research shows that spreadsheets often contain faults. For example, in an early spreadsheet study, 44%

of “finished” spreadsheets still contained faults [4]. A more recent survey of other such studies reported faults in 38% to 77% of spreadsheets at a similar stage [25]. Of perhaps greater concern, this survey also includes studies of “production” spreadsheets actually in use for day-to-day decision-making: from 10.7% to 90% of these spreadsheets contained faults.

One possible factor in this problem is the unwarranted confidence spreadsheet developers have in the reliability of their spreadsheets [10]. Another is the difficulty of creating and debugging spreadsheets: in interviews, experienced spreadsheet users reported that debugging spreadsheets could be hard because tracing long chains of formulas is difficult and because the effects of a small fault may not be visible until they have been propagated to a final result [14, 22].

To begin to address these problems, our previous work [30] presented a testing methodology for spreadsheets. That methodology allowed the user to indicate which cells are correct for a given test case, and to view test-ness information inferred from those marks. Building on that work, this paper describes our approach to integrating support for debugging and fault localization with that methodology. This integrated methodology adds the ability to mark which cells are *incorrect* for a given test case, and to view fault localization information inferred from both correct and incorrect marks. Key to the effectiveness of our approach is that it is tightly integrated into the spreadsheet environment, facilitating the incremental testing and debugging activities that normally occur during spreadsheet development. Our methodology also employs immediate visual feedback to present information in a manner that requires no knowledge of the underlying testing and fault localization theories.

2 Background: Testing Spreadsheets

The underlying assumption in our previous work has been that, as the user develops a spreadsheet incrementally, he or she is also testing incrementally. We have integrated a prototype implementation of our approach to incremental, visual testing into the spreadsheet language Forms/3 [6]; the examples in this paper are presented in that language.

Testing following our methodology [30] is intended for spreadsheet developers, not software engineers. Thus, our methodology does not include specialized testing vocabulary – in fact, it includes no vocabulary at all, instead presenting test-related information visually. Users test spreadsheets by trying different input values, and validating correct cells with a checkmark. Cells start out with red borders, indicating that they are untested. As cells are checked, their border colors change along a red-blue continuum, becoming bluer as the cell's testedness increases. When all the cells are blue, the spreadsheet is considered tested.

Although users of our methodology need not realize it, they are actually using a dataflow test adequacy criterion [18, 23, 26] and creating *du-adequate* test suites. In the theory that underlies this methodology, a *definition* is a point in the source code where a variable is assigned a value, and a *use* is a point where a variable's value is used. A *definition-use pair*, or *du-pair*, is a tuple consisting of a definition of a variable and a use of that variable. A *du-adequate* test suite is based on the notion of an *output-influencing all-definition-use-pairs-adequate test suite* [13] and is a test suite that exercises each *du-pair* in such a way that it participates (dynamically) in the production of an output explicitly validated by the user.

In spreadsheet terms, cells are considered variables. A cell is used when another cell references it, and a cell is defined within its own formula. If a cell's formula contains *if* expressions, then the cell can have multiple definitions. The testedness of a cell is calculated as the number of validated *du-pairs* with uses in that cell, divided by the total number of *du-pairs* with uses in that cell. Also, in the output-influencing scheme, testedness propagates against dataflow, so that if a cell *a* is validated, and if one of the *du-pairs* that provided *a*'s validated value has its definition in cell *b*, then any *du-pairs* that participated in providing *b*'s value are also considered tested.

This underlying theory is hidden from the user, for whom *du-pairs* represent interactions between cells

caused by references in cell formulas. These interactions can be visualized by the user through the display of dataflow arrows between subexpressions in cell formulas, and these arrows are colored to indicate whether the corresponding interaction has been tested.

This methodology also lets the user incrementally and simultaneously develop and test their spreadsheets. If the user adds a new formula or alters an existing formula, the underlying evaluation engine determines the *du-pairs* affected by this alteration and updates stored and displayed testing information. In this context, the problem of incremental testing of spreadsheets is similar to the problem of regression testing [29] and our solution emphasizes the importance of retesting code affected by modifications.

Figure 1 illustrates our prototype implementation of this methodology in use. The figure depicts a Forms/3 spreadsheet implementing a simple security check. Three key values identifying a person are placed in the cells *key1*, *key2*, and *key3*. The output cells *key1_out*, *key2_out*, and *key3_out* give a garbled version of the original keys that can be checked against a data base to determine if the person can be accepted. The spreadsheet developer initially validated the three output cells in this program. Then, to test further, the developer entered a different test case consisting of a different value for *key3*. Doing so changed the checkmark on *key3_out* to a question mark, indicating that previously displayed values have been validated, but the current ones have not. The formula for *key3_3* contains an *if* expression. So far only one branch of this expression has been tested, so the borders for the *key3_out* and *key3_3* cells are purple (gray in this paper). Cell *key3_2* has not been tested at all, so it is red (a light gray in this paper). Cells *key1_out*, *key1_1*, *key2_out*, *key2_1*, and *key3_1* have been completely tested, and have blue borders (black in this paper). The colors of displayed arrows between cells indicate the degree to which dependencies (interactions) between those cells have been validated.

3 An Integrated Methodology for Testing and Debugging Spreadsheets

During the course of a spreadsheet development session, users will locate failures in their spreadsheets: cases where cell outputs are incorrect. The interactive and incremental manner in which spreadsheets are created suggests that on discovering such failures, users may immediately attempt to locate and correct the faults that cause those failures.

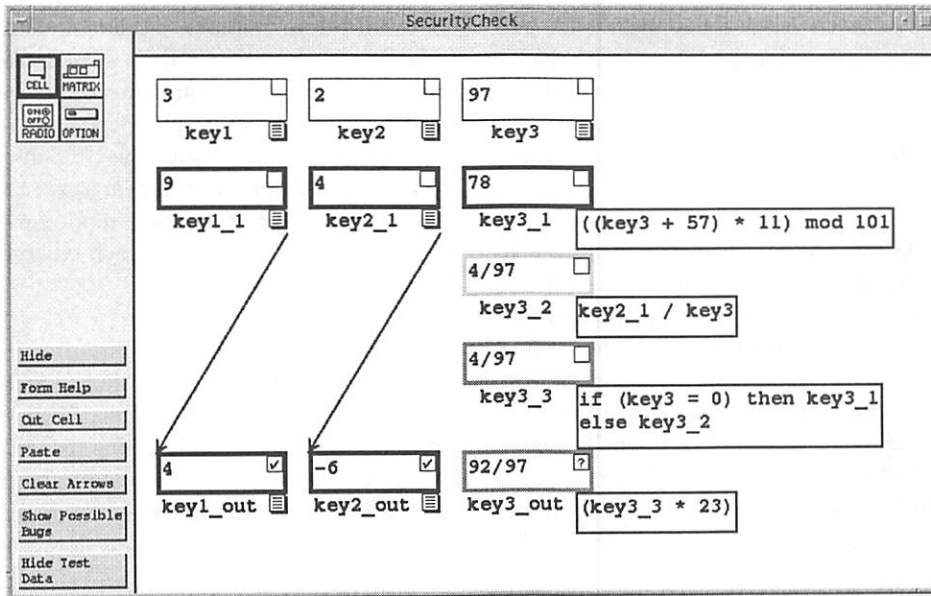


Figure 1: Forms/3 SecurityCheck spreadsheet with testing information displayed.

We wish to provide spreadsheet end users with automated support for this process of debugging and fault localization. There are three attributes of spreadsheet languages and their users that place constraints on methodologies providing such support:

- **Spreadsheets are modelless.** Spreadsheet creation does not require the separate code, compile, link, and execute modes typically required by traditional programming languages. Spreadsheet developers simply write formulas, enter values, and see results. Thus, in order for testing and debugging techniques to be useful for spreadsheet developers, the developers must be allowed to debug and test incrementally in parallel with spreadsheet development.
- **Spreadsheet developers are not likely to understand testing and debugging theory.** Given their end user audience, spreadsheet testing and debugging techniques cannot depend on the user understanding testing or debugging theory, nor should they rely on specialized vocabularies based on such theory.

Furthermore, spreadsheet developers are not liable to understand the reasons if debugging feedback leads them astray. They are more likely to become frustrated, lose trust in our methodology, and ignore the feedback. Therefore, our methodology must avoid giving false indications of faults where no faults exist.

- **Spreadsheets offer immediate feedback.** When a spreadsheet developer changes a formula, the spreadsheet displays the results quickly. Users have come to expect this responsiveness from spreadsheets and may not accept functionality that significantly inhibits responsiveness. Therefore the integration of testing and debugging into the spreadsheet environment must minimize the overhead it imposes.

Our methodology has been developed with these constraints in mind.

3.1 Slicing and dicing

Our debugging methodology is based on techniques for program slicing and dicing developed originally for imperative programs. We briefly review those techniques here in turn.

Program slicing was introduced by Weiser [34] as a technique for analyzing program dependencies. A program slice is defined with respect to a slicing criterion $\langle s, v \rangle$ in which s is a program point and v is a subset of program variables. A slice consists of a subset of program statements that affect, or are affected by, the values of variables in v at s [34]. *Backward slicing* finds all the statements that affect a given variable at a given statement, whereas *forward slicing* finds all the statements

that are affected by a given variable at a given statement. Weiser's slicing algorithm calculates *static* slices, based solely on information contained in source code, by iteratively solving dataflow equations. Other techniques [15, 24, 27, 31] calculate static slices by constructing and walking dependence graphs.

Korel and Laski [16] introduced *dynamic slicing*, in which information gathered during program execution is also used to compute slices. Whereas static slices find statements that may affect (or may be affected by) a given variable at a given point, dynamic slices find statements that may affect (or may be affected by) a given variable at a given point under a given execution. Dynamic slicing usually produces smaller slices than static slicing. Dynamic slices are calculated iteratively in [16]; an approach that uses program dependence graphs has also been suggested [1].

A great deal of additional work has been done on program slicing. An extensive survey of slicing is given in [32]. A more recent survey of dynamic slicing is given in [17].

Program dicing was introduced by Lyle and Weiser [20] as a fault localization technique for further reducing the number of statements that need to be examined to find faults. Whereas a slice makes use only of information on incorrect variables at failure points, a dice also makes use of information on correct variables, by subtracting the slices on correct variables away from the slice on the incorrect variable. The result is smaller than the slice on the incorrect variable; however, a dice may not always contain the fault that led to a failure.

Lyle and Weiser describe the cases in which a dice on an incorrect variable not caused by an omitted statement is guaranteed to contain the fault responsible for the incorrect value in the following theorem [20]:

Dicing Theorem. A dice on an incorrect variable contains a fault (except for cases where the incorrect value is caused by omission of a statement) if all of the following assumptions hold:

1. Testing has been reliable and all incorrectly computed variables have been identified.
2. If the computation of a variable, v , depends on the computation of another variable, w , then whenever w has an incorrect value then v does also.
3. There is exactly one fault in the program.

In this theorem, the first assumption eliminates the case where an incorrect variable is misidentified as a correct variable. The second assumption removes the case where a variable is correct despite depending on an incorrect variable (e.g. when a subsequent computation happens to compensate for an earlier incorrect computation, for certain inputs.) The third assumption removes the case where two faults counteract each other and result in an accidentally correct value.

Given the assumptions required for the Dicing Theorem to hold, it is clear that dicing must be an imperfect technique in practice. Thus, Chen and Cheung [8] explore strategies for minimizing the chance that dicing will fail to expose a fault that could have produced a particular failure, including the use of dynamic rather than static slicing.

3.2 Integrated testing and debugging

We have developed an integrated and incremental testing and debugging methodology that uses a fault localization technique similar to dicing. To achieve a close integration with the spreadsheet environment, our methodology gives spreadsheet developers the ability to edit, test, or debug a spreadsheet at any point during the development process without losing previously gathered testing or debugging information. To make this possible, our methodology provides the following user operations:

- The ability to view or hide testing and fault localization information at any time.
- The ability to incrementally mark cell values correct or incorrect for a single test case.
- The ability to change test cases without losing testing or debugging information gathered during previous testing.
- The ability to make a potential bug fix or other formula edit without losing testing or debugging information.

We next discuss how our methodology provides these functionalities while satisfying the constraints imposed by spreadsheet environments. We present the material in the context of an integrated spreadsheet development, testing, and debugging session.

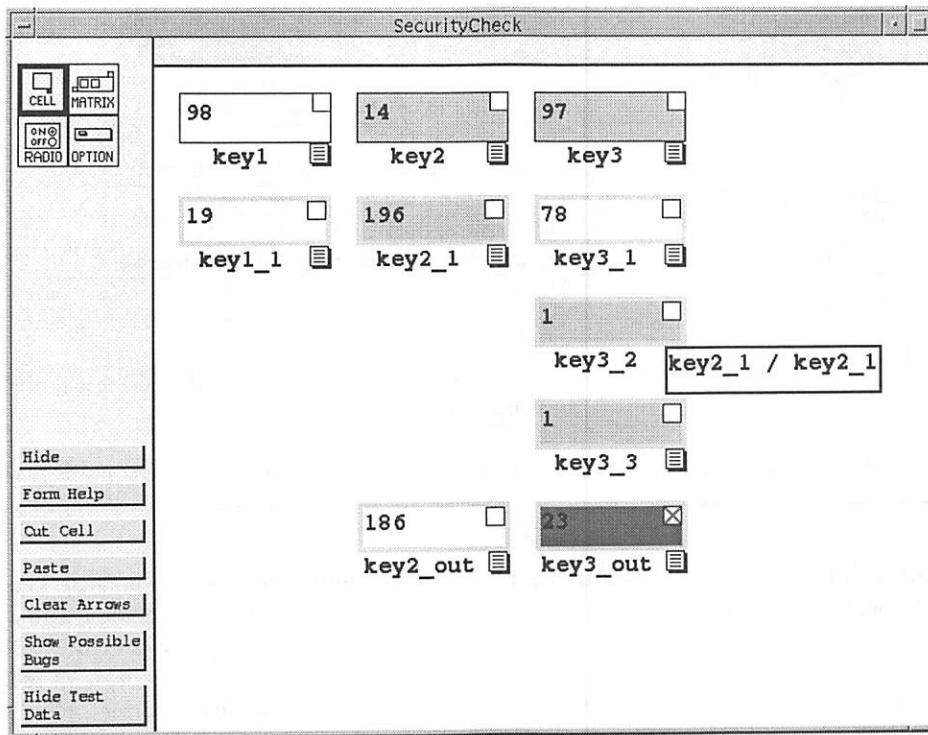


Figure 2: SecurityCheck spreadsheet at an early stage.

3.2.1 Spreadsheet development and simple fault localization

Suppose that, starting with an empty spreadsheet, the user begins to build the SecurityCheck application discussed in Section 2 and reaches the state shown in Figure 2. At this state, the user's spreadsheet contains an incorrect output: the *key3_out* cell, which should contain the value (4508/97), contains the value 23. This is caused by an incorrect cell reference in the cell *key3_2*. Instead of dividing the value of *key2_1* by the value of *key3*, the formula for *key3_2* divides *key2_1* by itself.

As soon as an incorrect output is noticed, users can place an "X" mark in a cell to indicate that it has an incorrect value. Suppose the user places such a mark in the *key3_out* cell.

Now, suppose the user decides to investigate the cause of this failure immediately. Having placed one or more X marks, the user can view fault localization information by pressing a "Show Possible Bugs" button. This causes cells suspected of containing faults to be highlighted in red (gray in this paper); in this case, the highlighted cells are those contained in the backward dynamic slice

of *key3_out*. Note that the border color scheme is different from that used in our previous testing methodology. Previously border colors went from red, representing untested, to blue, representing tested. Now border colors start out at black (light gray in this paper) to represent untested, move to various shades of purple to represent partially tested, and finally move to blue (black in this paper) to represent fully tested. This color scheme was chosen to avoid a conflict between red border colors representing an untested cell, and a red background color representing a potentially faulty cell.

Now six of the cells are highlighted red, including *key3_out*. Two of these cells are constant cells. For our purposes, a constant cell is any cell whose formula does not refer to another cell. These cells are highlighted in case the incorrect value is caused by a data entry error. At this point the user, knowing that the entered data is correct, can ignore those cells and concentrate on the remaining four cells.

$Predecessors(C)$	The set of cells in S that C references in its formula.
$Successors(C)$	The set of cells in S that reference C in their formulas.
$DynamicPredecessors(C)$	The set of cells $D \in S$ such that D 's value was used the last time the value of C was computed.
$DynamicSuccessors(C)$	The set of cells $D \in S$ such that D used the value of C the most recent time D 's value was computed.
$BackwardSlice(C)$	The transitive closure on $Predecessors(C)$.
$ForwardSlice(C)$	The transitive closure on $Successors(C)$.
$DynamicBackwardSlice(C)$	The transitive closure on $DynamicPredecessors(C)$.
$DynamicForwardSlice(C)$	The transitive closure on $DynamicSuccessors(C)$.
$IncorrectDependentsOf(C)$	The set of cells in $DynamicForwardSlice(C)$ that have been marked incorrect.
$CorrectDependentsOf(C)$	The set of cells in $DynamicForwardSlice(C)$ that have been marked correct.

Table 1: The definitions used in determining fault likelihood. Here S is a spreadsheet, and C is any cell in S .

3.2.2 Applying additional knowledge to further refine fault localization information

Dicing in a spreadsheet environment would find the set of cells that contribute to a value marked incorrect but not to a value marked correct. The set of cells indicated by dicing could exclude a fault if one of the conditions in Dicing Theorem were violated. However, one constraint our methodology must satisfy is that the user should not be frustrated by searching through highlighted cells to find that none of them contain faults. We believe that the restrictions imposed by the Dicing Theorem are too strict to be practical in a spreadsheet environment. Therefore dicing cannot be used for our methodology.

Dicing makes a binary decision about cells: either a cell is indicated or it is not. To allow the conditions in the Dicing Theorem to be violated without causing user frustration, our technique does not make a binary decision about which cells to include or exclude. Instead, our methodology estimates the likelihood that a cell contributes to a value marked incorrect. This likelihood is presented to the spreadsheet developer by highlighting suspect cells in different shades of red. We call this likelihood the *fault likelihood* of a cell. Let I be the set of cell values marked incorrect by the spreadsheet developer. The fault likelihood of a cell C is an estimate of the likelihood that C contains a fault that contributes to an incorrect value in I .

Returning to the `SecurityCheck` example, suppose that having seen several cells highlighted in red as potentially faulty, the user does not yet wish to examine formulas, but would prefer to restrict the potential fault site further. One way to do so is to test other parts of the spreadsheet for correctness. In this case, only one of the

other output cells has been created, `key2.out`. This cell is correct, so the user can mark it with a check box. The result of doing so is shown in Figure 3. Now both `key2.1` and `key2` contain a lighter shade of red than before because they contribute to a correct cell value. The shade of red in the background of a cell indicates the fault likelihood of that cell. A lighter shade of red indicates a lower likelihood, and a darker shade of red indicates a higher likelihood.

There is no way to compute an exact value for the fault likelihood of a cell: we can only estimate it based on the number of values marked correct or incorrect that depend on a cell's value. Our strategy for doing so is to maintain six properties, described below, which rely on the definitions in Table 1.

Property 1 *If $IncorrectDependentsOf(C) \neq \emptyset$ then C has at least a minimal fault likelihood.*

This property ensures that every cell in the backward dynamic slice of a value marked incorrect will be highlighted. This reduces the chance that the user will become frustrated searching for a fault that is not there, and in our opinion is essential to a fault localization methodology for spreadsheets. However, there are still two situations in which the highlighted cells might not include a fault responsible for a value marked incorrect. The first situation can occur when a fault is caused by the omission of a cell. The second situation can occur when a correct value is mistakenly marked incorrect. These situations, however, cannot in general be avoided by any fault localization methodology.

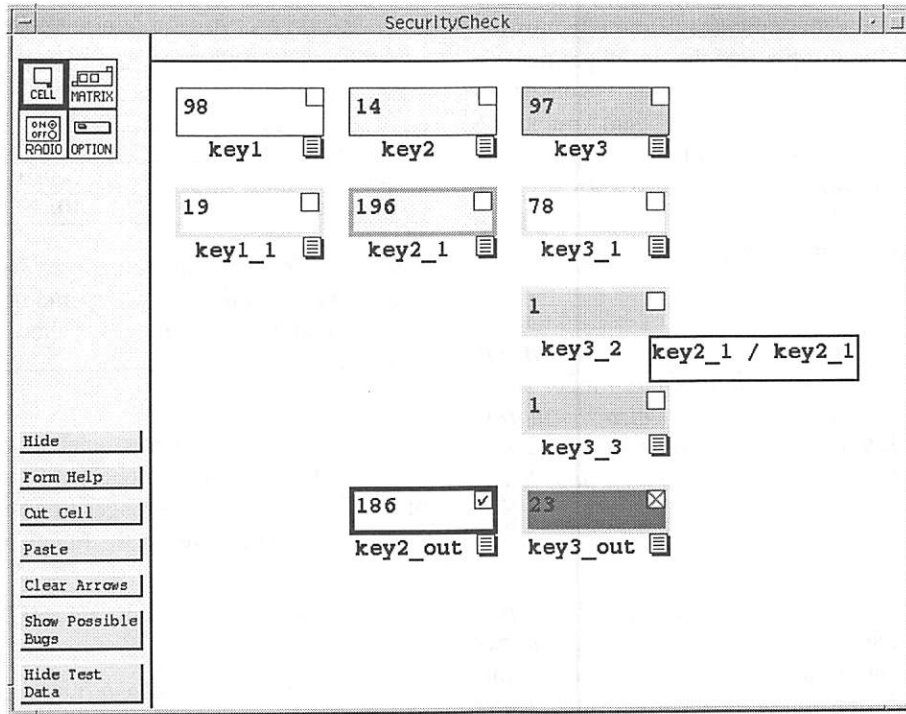


Figure 3: SecurityCheck spreadsheet following additional validation.

Property 1 determines the overhead imposed by the actions of marking a cell correct or incorrect. The time complexity of both operations must be $O(|DynamicBackwardSlice(C)|)$, where C is the cell being marked. This is approximately the same as the cost of computing the value of C for the first time (i.e., when the predecessors also need to be computed).

In order to localize a fault to a set of cells smaller than the dynamic backward slice, we maintain several other properties to determine how fault likelihood should be estimated. These properties ensure that cells highlighted a bright shade of red have a higher likelihood of containing a fault than those marked a lighter shade of red.

Property 2 *The fault likelihood of C is proportional to $|IncorrectDependentsOf(C)|$.*

Property 3 *The fault likelihood of C is inversely proportional to $|CorrectDependentsOf(C)|$.*

Property 2 is based on the assumption that the more incorrect computations a cell contributes to, the more likely it is that the cell contains a fault. Conversely, Property 3

is based on the assumption that the more correct computations a cell contributes to, the less likely it is that the cell contains a fault.

Property 4 *If C has a value marked incorrect, the fault likelihood of C is high.*

This property is based on the assumption that a good place to start looking for a fault is the place where the effect of the fault was first recognized. We give that cell a high fault likelihood so that it stands out.

Property 5 *An incorrect mark on C blocks the effects of any correct marks on cells in $DynamicForwardSlice(C)$, preventing propagation of the correct marks' effects to the fault likelihood of cells in $DynamicBackwardSlice(C)$.*

This property is relevant when a correct cell value depends on an incorrect cell value. There are three possible explanations for such an occurrence. The first is that a formula of one of the cells between the correct cell and the incorrect cell somehow converts the incorrect value to a correct one. The second is that there is another fault

between the two cells that counteracts the effect of the incorrect value. The third is that the developer made a mistake in marking one of these two cells. We choose to trust the developer's decision in this case and assume one of the first two situations. For both of these situations the incorrect value does not contribute to the correct value. Therefore the effects of the correct mark should not propagate back to cells the incorrect value depends on.

Property 6 *A correct mark on C blocks the effects of any incorrect marks on cells in $\text{DynamicForwardSlice}(C)$, preventing propagation of the incorrect marks' effects to the fault likelihood of cells in $\text{DynamicBackwardSlice}(C)$, except for the minimal fault likelihood required by Property 1.*

This property is relevant when a value marked incorrect depends on a value marked correct. In dicing, the dynamic backward slice of the correct value would be completely subtracted from that of the incorrect value. However we need to be more conservative and assume that a violation of the Dicing Theorem is possible. Thus, the cells in the dynamic backward slice of the correct value are given a low but nonzero fault likelihood.

3.2.3 Implementing the Properties

The above properties allow for many different ways of estimating fault likelihood. First we require the following definitions to handle the "blocks" introduced by the fifth and sixth properties. Let $\text{NumBlockedIncorrectDependentsOf}(C)$ be the number of cell values belonging to cells in $\text{DynamicForwardSlice}(C)$ that are marked incorrect but are blocked by a value marked correct along the data flow path from C to the value marked incorrect. Let $\text{NumReachableIncorrectDependentsOf}(C)$ be $|\text{IncorrectDependentsOf}(C)| - \text{NumBlockedIncorrectDependentsOf}(C)$, or in other words the number of cell values marked incorrect whose effects reach C without being blocked. Similar definitions are made for $\text{NumBlockedCorrectDependentsOf}(C)$ and $\text{NumReachableCorrectDependentsOf}(C)$.

As a starting point, we decided to divide fault likelihood into six distinct ranges: "none", "very low", "low", "medium", "high", and "very high". To estimate the fault likelihood for a cell C , we first assign a range to the values of $\text{NumReachableIncorrectDependentsOf}(C)$ and $\text{NumReachableCorrectDependentsOf}(C)$ using Table

Range	$\text{NumReachableIncorrectDependentsOf}(C)$ or $\text{NumReachableCorrectDependentsOf}(C)$
none	0
low	1-2
medium	3-4
high	5-9
very high	10+

Table 2: $\text{NumReachableIncorrectDependentsOf}(C)$ and $\text{NumReachableCorrectDependentsOf}(C)$ yield six distinct fault likelihood ranges as shown.

2. These ranges can be associated with numeric values from 0, representing "none", to 5, representing "very high". To combine these ranges to determine the resulting fault likelihood, we use the function:

$$\text{fault likelihood}(C) = \max(1, RID - \left\lfloor \frac{RC}{2} \right\rfloor)$$

where $RID = \text{NumReachableIncorrectDependentsOf}(C)$ and $RC = \text{NumReachableCorrectDependentsOf}(C)$.

We make three exceptions. If $\text{IncorrectDependentsOf}(C) = \phi$, then the fault likelihood of C is "none". If C has been marked incorrect, its fault likelihood is assigned a value of "very high", in keeping with Property 4. If $\text{NumReachableIncorrectDependentsOf}(C) = 0$, but $\text{NumBlockedIncorrectDependentsOf}(C) > 0$, then the fault likelihood of C is assigned a value of "very low". This is to maintain Property 1, and ensures that every cell in the dynamic backward slice of a value marked incorrect is highlighted.

Returning to Figure 3, *key3.out* is a bright red color because it has a "very high" fault likelihood. Cells *key3.3* and *key3.2* each have one reachable incorrect dependent, so they have fault likelihoods of "low". Cell *key2.1* has a "very low" fault likelihood because it has one reachable incorrect dependent and another reachable correct dependent.

3.2.4 Applying additional test cases

Suppose the user developing the Security Check spreadsheet still wants to further narrow down the set of possible locations of the fault. One option is for the user to apply additional test cases. Figure 4 shows the result of entering a new test case into *key1*, *key2*, and *key3*. Now both *key2.out* and *key3.out* are correct, so the user checks both cells. The information about the previous test case is not lost, so now *key3.3* has a reach-

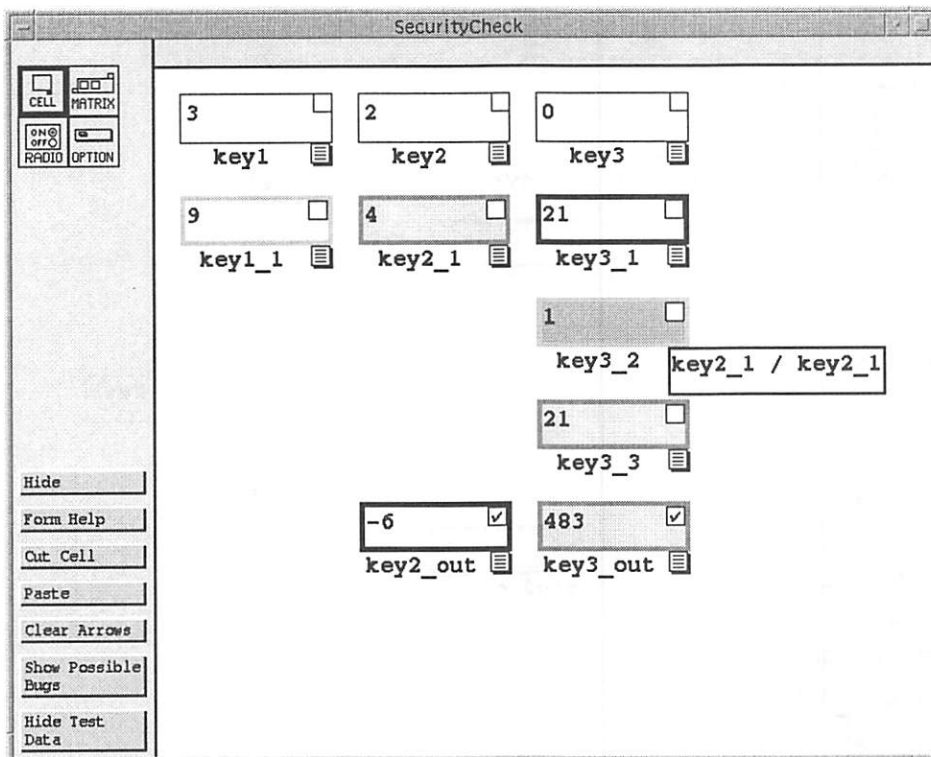


Figure 4: SecurityCheck spreadsheet following application of additional test cases.

able correct dependent for this test case and a reachable incorrect dependent for the previous test case. This gives *key3.3* a fault likelihood of “very low”. However, in this test case *key3.2* is no longer in the dynamic backward slice of *key3.out*. This is because *key3.3* is designed to not use *key3.2* if its result would be a divide by zero. There is still one reachable incorrect dependent from the previous test case for *key3.2*, so its fault likelihood stays “low”. Now the faulty cell, *key3.2*, has the brightest red color on the form, suggesting that it is most likely to contain a fault.

In order for testing and debugging information to be preserved between test cases, our methodology must respond correctly to formula edits. Any formula edit that changes a constant cell to another constant is considered a change in test case. When this occurs, all of the marked cells dependent on the changed cell must have their marks removed and replaced with question marks. However, the effects of those marks on cell colors and testing and debugging information should not be removed by changing a test case. In effect, the mark is “hidden” behind the question mark.

This process of preserving testing and debugging information across edits adds no additional time complexity to the process of editing a cell, because the spreadsheet environment must already visit the dependents of a changed cell in order to mark them as requiring recomputation.¹

3.2.5 Maintaining testing and fault information after changes to formulas

Now suppose the developer of the Security Check application decides to fix the fault. This involves editing the formula for *key3.2* from “*key2.1/key2.1*”, to “*key2.1/key3*”. Figure 5 shows the result of this action. As expected, *key3.2* now contains a divide by zero error; this is why *key3.3* uses *key3.1* instead. However, now that the formula has changed, the marks placed on *key3.out* are out of date. Not only must they be removed, but their effects on testing and debugging information must be undone. In effect, the affected cells can

¹For more information on marking and evaluation strategies applicable to spreadsheet languages, see[5].

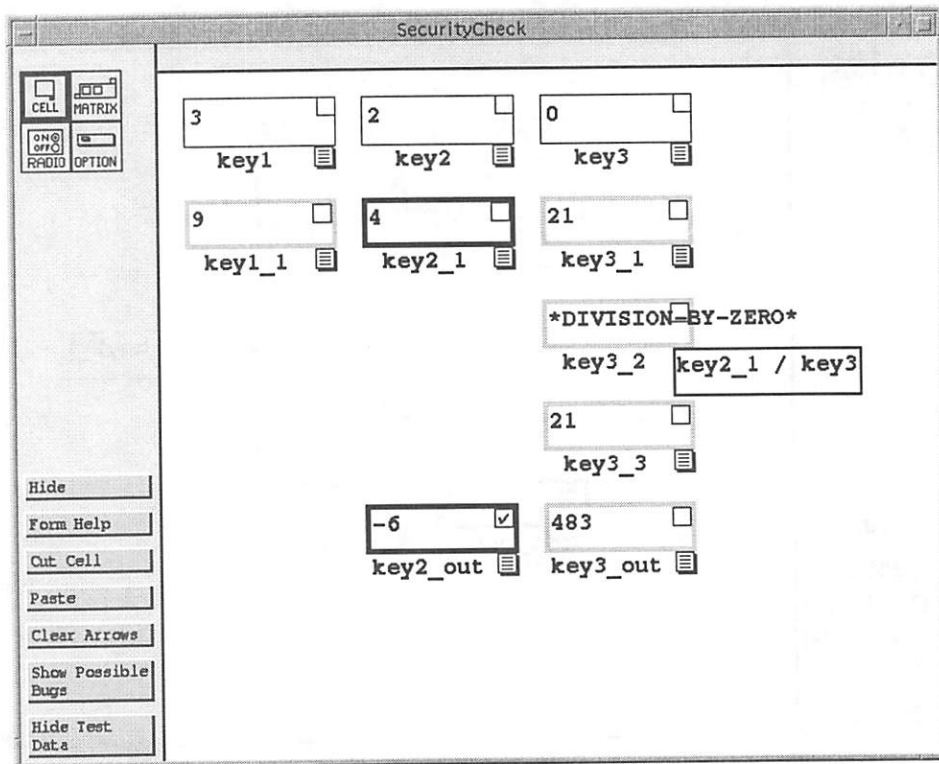


Figure 5: Corrected SecurityCheck spreadsheet.

no longer be considered tested because they rely on a new and untested formula. This also encourages the user to perform regression testing on the cells affected by the change.

Whenever a formula edit is made that does not change a constant to a constant, testing and debugging information must be removed. This information must be removed for all prior test cases, so the effects of every mark ever placed on a cell in the static forward slice of the edited cell must be removed.

Furthermore, our methodology relies on static and dynamic slicing information. This information must be kept up to date whenever any edit is made.

Whenever a formula edit occurs, the spreadsheet environment must mark all affected cells to be recomputed. This requires storing some form of static or dynamic successor information. An algorithm for a formula edit that maintains static successor information for a cell C must visit the cells in $Predecessors(C)$ and update their static successor information. This includes the cells in $Predecessors(C)$ both before the formula change and af-

ter the change. The static predecessor information can be gathered from the cell's formula. Let SP represent the the maximum number of static predecessors before and after the formula edit. The worst case time complexity for such an algorithm is $O(SP + |ForwardSlice(C)|)$. Such an algorithm provides all the information needed for static slicing; the slicing algorithm can perform a graph walk using the cell's formula during a backward slice, and the static successor information for a forward slice.

Dynamic successor and predecessor information can be maintained during the process of recalculating cells. An algorithm to recompute a cell C and maintain dynamic successor and predecessor information must first visit the cells in $DynamicPredecessors(C)$ to update their dynamic successor information. Then C must be recomputed. This involves visiting every cell required for C 's new value. During this process, the dynamic predecessor information for C can be collected, and the dynamic successor information for the cells used to compute C can be updated. Thus the only additional cost needed to maintain dynamic slicing information is the time needed to update the dynamic successor infor-

mation for the cells previously used by C . Let DP be the maximum number of dynamic predecessors of C , both before and after C is recomputed. The time complexity for an algorithm to compute C 's value is $O(DP + COMP)$, where $COMP$ is the time required to compute the cell's value once the values of its predecessors are known. Note that in some spreadsheet languages cell formulas can use functions whose time complexity is not bounded by the number of operands. However in order to evaluate the effect our methodology has on responsiveness, we assume the worst case scenario in which the time complexity of recomputing a cell is dominated by the cost of maintaining dynamic successor and predecessor information.

Our methodology also adds overhead by removing testing information when editing a formula. As discussed earlier, a formula edit to provide a different test case adds no additional overhead to an algorithm for accepting an edit to a cell. However, editing a non-constant formula does. An algorithm that handles such an edit to a cell C while maintaining static slicing information and testing and debugging information must not only visit the cells in $ForwardSlice(C)$, but also the cells in the backward static slices of every cell that had been marked during the current or a previous test case. Let M be the number of cells marked in $ForwardSlice(C)$. Let B be the maximum length of a backward static slice of a marked cell in $ForwardSlice(C)$. The worst case time complexity for editing a formula while maintaining fault localization information is $O(SP + M \cdot B + |DynamicForwardSlice(C)|)$.

To analyze the effects on responsiveness of maintaining testing and debugging information, we must consider two separate activities. The first is the formula edit itself. The second is the calculation phase where at minimum every cell on the screen is recalculated. From the user's point of view, the system is responsive only if both the edit and the recalculation of on-screen cells happen quickly. Therefore we merge the two and define a *responsive edit* to be an operation where a cell is edited and at least the on-screen cells are updated. For our methodology, the worst case time complexity for a responsive edit is $O(SP + M \cdot B + DP \cdot |DynamicForwardSlice(C)|)$. There are three overhead factors that must be considered, SP , DP , and $M \cdot B$.

Spreadsheet environments maintain successor information in order to mark or recompute cells that need to be recomputed. Therefore, either SP or DP will already be a factor in the complexity for a spreadsheet environment that does not implement our methodology. However, one of these factors may be added as overhead

from our methodology. The factor DP is likely to have a larger effect than SP , because every cell that is recalculated to update the screen must update the dynamic successor information of its predecessors. In spreadsheet languages that do not support ranges, such as Forms/3, DP or SP are likely to be constant bound. This is because the number of predecessors depends on the number of references in a formula. Since long formulas are often awkward, users are more likely to break them up into distinct cells than to create large formulas that reference a large number of cells. However, if ranges are added, a short formula is capable of referencing a large number of cells. Ranges also add additional complexity for visualizing a slice, as drawing an arrow to each cell in the range will result in a jumble of arrows. This can be dealt with by handling a range as a single entity. For example, the data flow arrows in Excel point to a box around a range rather than to every cell in the range.

The largest factor introduced by our methodology is likely to be the $M \cdot B$ factor introduced when changing a formula. Since B is the size of a static slice, it could be large. However we have no way of knowing how large M will be, as this depends on how many marks are placed by the user.

It remains to be seen if the factors of $M \cdot B$, SP or DP can be large enough to negatively affect the responsiveness of our methodology. However, there are approaches that can mitigate the effects of these factors. One approach would be a multithreaded approach to updating cell values and testing and debugging information. A background thread can be used to update information while the user is free to continue interacting with the spreadsheet. This allows the spreadsheet environment to stay interactive. However as the user makes changes the state of the spreadsheet could become increasingly inconsistent. This can be partially mitigated by placing priority on updating information for on-screen cells. This approach would be useful only if the benefits to be gained by our technique outweighed the difficulty of implementing it.

4 Related Work

Previous work related to testing and debugging spreadsheets has been limited to auditing tools. Most of the work on such tools has taken place in commercial applications. For example, Microsoft Excel 97 allows users to place restrictions on the value of a cell, to place comments on cells, and to draw arrows that track dataflow dependencies. The dataflow arrows implement a form of

static slicing that lets users view backward and forward slices by expanding the arrows out one dependence level at a time.

The only research work we are aware of addressing auditing tools is by Davis [12]. Davis proposes two tools, a dataflow arrow tool and an automatically derived dataflow graph. The arrow tool is similar to the arrows in Excel; however, from any one cell it is possible only to request arrows that show one dependence level. The dataflow graph tool draws a graph using different symbols to categorize cells as input, output, decision variables, parameters, or formulas. This graph is similar to the graph suggested by Ronen, Palley and Lucas [28], however it is generated automatically by the system instead of being drawn by the user.

Our testing and debugging methodology utilizes dataflow arrows similar to those used by Excel and Davis's tools. Unlike Excel arrows, however, our dataflow arrows allow users to display an entire slice at once, or to limit the depth of the slice. The user can also choose between forward and backward slicing, as well as dynamic and static slicing. At the testing level, users can also display dataflow arrows at a finer granularity (between subexpressions); this ability will soon be adapted into our debugging functionality.

There has also been research into using interactive, visual techniques to aid in debugging, particularly as it relates to program comprehension (eg: [2, 19]). This approach integrates debugging functionality with reviewing execution histories, both graphically and textually, to better understand program behavior and thus find faults.

5 Conclusions and Future Work

Due to the popularity of commercial spreadsheets, spreadsheet languages are being used to produce software that influences important decisions. Furthermore, due to recent advances from the research community that expand its capabilities, the use of this paradigm is likely to continue to grow. We believe that the fact that such a widely-used and growing class of software often has faults should not be taken lightly.

To address this issue, we have developed a methodology that brings some of the benefits of formal testing and debugging methodologies to this class of software. Our methodology is tightly integrated into the spreadsheet environment, facilitating the incremental testing and debugging activities that occur during spreadsheet devel-

opment. Our methodology also employs the visual feedback that is characteristic of spreadsheet environments, while presenting information in a manner that requires no knowledge of the underlying testing and fault localization theories.

Future work is planned along two dimensions. First, although our previous user studies have suggested that visual feedback about testing coverage helps users correct faults [11], we have not yet conducted a user study involving the methodology presented here. We are currently designing such a study.

Second, we expect user studies to reveal possibilities for new or refined debugging techniques. For example, we may wish to investigate the use of slicing and fault localization at the level of subexpressions: this may yield more precise results, but at additional cost. Another possible direction is to refine the user interface for our methodology. Our current interface limits the user to only the information that can be displayed on one screen. A more scalable approach, such as the one used in [3] for C programs, would allow users to scan through slicing and fault localization information for a large spreadsheet program without having to scroll within or switch between multiple worksheets.

Although our current research is with spreadsheet languages, we believe this methodology could be extended to other end user languages. The notion of fault likelihood used in this paper could be used in other environments in which slicing is available. However, availability of slicing is not sufficient for our methodology. In addition the slicing must be highly accurate, to avoid giving false indications that frustrate the user. Furthermore the environment must also support interactive editing of source code, editing of test cases, and validation of output, and must provide immediate feedback as to the effects of those actions. This combination of requirements suggests that our methodology is best suited for the highly interactive visual environments that have recently begun to emerge for end user programming.

The studies presented in [14, 22] showed that end users find that the act of debugging, and particularly the act of locating faults in long computation chains, is very difficult in spreadsheet programs. Our methodology attempts to alleviate this difficulty by providing the user with slicing and fault localization information. The goal of this work is to provide effective testing and debugging methodologies that help reduce the number of faults in spreadsheet programs and may also be helpful in other end user programming environments.

Acknowledgements

We thank the Visual Programming Research Group for their work on the Forms/3 implementation and for their feedback on the methodology. This work has been supported by NSF under ASC 93-08649, by NSF Young Investigator Award CCR-9457473, by Faculty Early CAREER Award CCR-9703108, and by ESS Award CCR-9806821 to Oregon State University.

References

- [1] H. Agrawal and J.R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 246–256, June 1990.
- [2] J. Atwood, M. Burnett, R. Walpole, Wilcox E., and S. Yang. Steering programs via time travel. In *IEEE Symposium on Visual Languages*, September 1996.
- [3] T. Ball and S.G. Eick. Visualizing program slices. In *Proceedings. IEEE Symposium on Visual Languages*, pages 288–95, October 1994.
- [4] P. Brown and J. Gould. Experimental study of people creating spreadsheets. *ACM Transactions on Office Information Systems*, 5(3):258–272, July 1987.
- [5] M. Burnett, J. Atwood, and Z. Welch. Implementing level 4 liveness in declarative visual programming languages. In *1998 IEEE Symposium on Visual Languages*, September 1998.
- [6] M. Burnett and H. Gottfried. Graphical definitions: Expanding spreadsheet languages through direct manipulation and gestures. *ACM Transactions on Computer-Human Interaction*, 5(1):1–33, March 1998.
- [7] M. Burnett, R. Hossli, T. Pulliam, B. VanVoorst, and X. Yang. Toward visual programming languages for steering in scientific visualization: a taxonomy. *IEEE Computer Science and Engineering*, 1(4), 1994.
- [8] T. Y. Chen and Y. Y. Cheung. On program dicing. *Software Maintenance: Research and Practice*, 9(1):33–46, January–February 1997.
- [9] E. H. Chi, P. Barry, J. Riedl, and J. Konstan. A spreadsheet approach to information visualization. In *IEEE Symposium on Information Visualization*, October 1997.
- [10] C. Cook, M. Burnett, and D. Boom. A bug's eye view of immediate visual feedback in direct-manipulation programming systems. In *Proceedings of Empirical Studies of Programmers*, October 1997.
- [11] C. Cook, K. Rothermel, M. Burnett, T. Adams, G. Rothermel, A. Sheretov, F. Cort, and J. Reichwein. Does immediate visual feedback about testing aid debugging in spreadsheet languages. Technical Report TR 99-60-07, Oregon State University, March 1999.
- [12] J.S. Davis. Tools for spreadsheet auditing. *International Journal of Human-Computer Studies*, 45(4):429–442, 1996.
- [13] E. Duesterwald, R. Gupta, and M. L. Soffa. Rigorous data flow testing through output influences. In *Proceedings of the 2nd Irvine Software Symposium*, March 1992.
- [14] D. G. Hendry and T. R. G. Green. Creating, comprehending and explaining spreadsheets: a cognitive interpretation of what discretionary users think of the spreadsheet model. *International Journal of Human-Computer Studies*, 40(6):1033–1065, June 1994.
- [15] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.
- [16] B. Korel and J. Laski. Dynamic slicing of computer programs. *The Journal of Systems and Software*, 13(3):187–195, November 1990.
- [17] B. Korel and J. Rilling. Dynamic program slicing methods. *Information and Software Technology*, 40(11–12):647–659, December 1998.
- [18] J. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, 9(3):347–354, May 1993.
- [19] H. Lieberman and C. Fry. Zstep 95: A reversible, animated source code stepper. In J. Stasko, J. Domingue, M. Brown, and B. Price, editors, *Software Visualization: Programming as a Multimedia Experience*. MIT Press, Cambridge, MA, 1997.
- [20] J.R. Lyle and M. Weiser. Automatic program bug location by program slicing. In *Proceedings of the 2nd International Conference, Computers and Applications*, pages 877–883, 1987.
- [21] B. Myers. Graphical techniques in a spreadsheet for specifying user interfaces. In *ACM CHI '91*, pages 243–249, April 1991.
- [22] B. A. Nardi and J. R. Miller. Twinkling lights and nested loops: distributed problem solving and spreadsheet development. *International Journal of Man-Machine Studies*, 34(2):161–184, February 1991.
- [23] S. C. Ntafos. On required element testing. *IEEE Transactions on Software Engineering*, 10(6), November 1984.
- [24] K.J. Ottenstein and L.M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–84, April 1984.
- [25] R. Panko and R. Halverson. Spreadsheets on trial: A survey of research on spreadsheet risks. In *Twenty-Ninth Hawaii International Conference on System Sciences*, January 1996.
- [26] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, April 1985.

- [27] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 11–20, December 1994.
- [28] B. Ronen, M.A. Palley, and H.C. Lucas. Spreadsheet analysis and design. *Communications of the ACM*, 32(1):84–93, January 1989.
- [29] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8), August 1996.
- [30] G. Rothermel, L. Li, C. Dupuis, and M. Burnett. What you see is what you test: A methodology for testing form-based visual programs. In *Proceedings of the 20th International Conference on Software Engineering*, pages 198–207, April 1998.
- [31] S. Sinha, M.J. Harrold, and G. Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Proceedings of the 21st International Conference on Software Engineering*, May 1999.
- [32] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.
- [33] G. Viehstaedt and A. Ambler. Visual representation and manipulation of matrices. *Journal of Visual Languages and Computing*, 3(3):273–298, September 1992.
- [34] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

An Annotation Language for Optimizing Software Libraries*

Samuel Z. Guyer

The University of Texas at Austin

sammy@cs.utexas.edu, <http://www.cs.utexas.edu/users/sammy>

Calvin Lin

The University of Texas at Austin

lin@cs.utexas.edu, <http://www.cs.utexas.edu/users/lin>

Abstract

This paper introduces an annotation language and a compiler that together can customize a library implementation for specific application needs. Our approach is distinguished by its ability to exploit high level, domain-specific information in the customization process. In particular, the annotations provide semantic information that enables our compiler to analyze and optimize library operations as if they were primitives of a domain-specific language. Thus, our approach yields many of the performance benefits of domain-specific languages, without the effort of developing a new compiler for each domain.

This paper presents the annotation language, describes its role in optimization, and illustrates the benefits of the overall approach. Using a partially implemented compiler, we show how our system can significantly improve the performance of two applications written using the PLAPACK parallel linear algebra library.

1 Introduction

Software libraries are a common mechanism for re-using code. Like a domain-specific language, libraries can provide high-level abstractions that empower the programmer and hide implementation details. Unlike a domain-specific language, libraries do not introduce new syntax and receive no direct support from the compiler. These differences have two consequences:

- **Compilers have limited ability to improve performance.** Compilers cannot exploit the domain-specific information that is only implicitly encoded in a library's implementation. Thus, many opportunities for optimization are lost. Since library code is written, compiled and optimized in isolation, such optimizations are important as a means of customizing a library implementation for different application needs.
- **Performance improvements are exposed through the interface.** The only way to offer both generality and performance is to provide wide interfaces with specialized routines for different contexts. Unfortunately, these specialized routines are typically more difficult to use correctly. Moreover, the specialized routines typically improve performance by exposing implementation decisions. Thus, they intertwine the interface and the implementation, which inhibits code reuse in the long run.

Our approach to mitigating these problems is to give libraries some of the compiler support enjoyed by domain-specific languages. The key is an annotation language that captures expert knowledge about libraries and enables our compiler to customize library implementations for different situations. Library users can then focus on application design, relying on our compiler to optimize performance.

Figure 1 shows the overall architecture of our system. The annotations are supplied by a library expert in a separate specification file that accompanies the usual header files and source code. The annotations convey two kinds of information about library routines: (1) basic dataflow information, which is sometimes difficult to obtain through static analysis, and (2) high level domain-specific information. Our compiler, which we

*This work was supported in part by an NSF Research Infrastructure Award CDA-9624082, NSF grant CCR-9707056, and ONR grant N00014-99-1-0402.

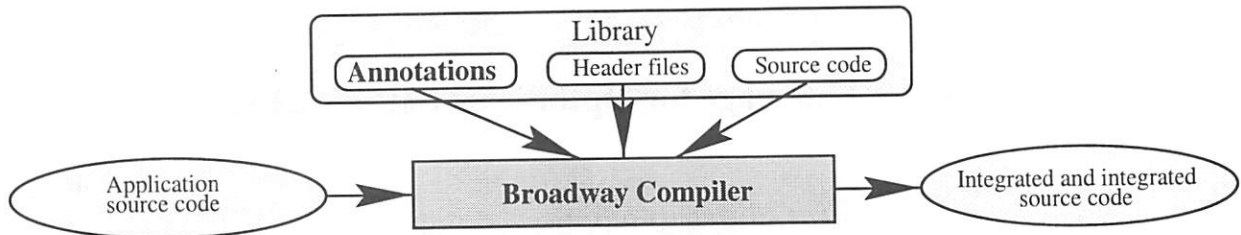


Figure 1: Architecture of the Broadway Compiler system

have named the Broadway compiler, reads the annotations and applies a series of source-to-source transformations to both the library and application source. The result is an integrated system of library and application code, which is ready to be compiled and linked using conventional tools.

Our system offers many practical benefits. First, the annotations are specified in a separate file from the library source, so our approach applies to existing libraries and existing applications. Second, the annotations describe the library, not the application, so the application programmer does nothing more than use the Broadway Compiler in place of a standard C compiler. Finally, the non-trivial cost of writing the library annotations can be amortized over many applications.

When applied to a Cholesky factorization program that uses the PLAPACK parallel linear algebra library [25], our system improves performance by 26% for large matrices and 195% for small matrices. Both the library and application are written in ANSI C, and neither has been modified to facilitate our results. This paper will explain how our solution is able to obtain these results. While these same optimizations could be performed manually by using a wider interface and expert knowledge of the PLAPACK implementation, our approach offers significant advantages:

- Both approaches require semantic expertise about the PLAPACK implementation, but manual optimization embeds this knowledge implicitly in the optimized program, while our annotations encapsulate such knowledge for use in optimizing other PLAPACK applications.
- Manual optimization is feasible only for PLAPACK experts. By contrast, once an expert has provided annotations, even casual users can optimize their PLAPACK applications by invoking our compiler.
- Manual optimization directly modifies the source code, which complicates subsequent modification,

reuse and maintenance. Our annotations instead provide a clean separation of the optimization information from the basic implementation.

- The explicit representation of semantic information allows it to be checked for correctness. This is an open issue which we leave as future work.

The performance improvements mentioned above cannot be obtained with conventional compiler technology because the optimizations require semantic information about the PLAPACK implementation that cannot be derived automatically. For example, one transformation requires knowledge of a PLAPACK object's data distribution. This information is implicitly represented in the values of four object attributes and the value of a global variable. To further complicate matters, PLAPACK is written in C and is difficult to analyze because of its pervasive use of pointers. In addition, certain def/use information is impossible to obtain because PLAPACK makes calls to the sequential BLAS library [11], whose source code is unavailable.

The primary contributions of this paper are (1) the introduction of a new technique for optimizing software libraries, (2) the demonstration that this technique provides performance benefits when applied to a production-quality library, and (3) an evaluation of our annotation language based on experiments with PLAPACK applications.

This paper is organized as follows. Section 2 contrasts our work with related efforts. Section 3 describes our annotation language and its design philosophy. Section 4 explains our compilation strategy, and Section 5 offers an empirical evaluation of our language. Finally, we draw conclusions and discuss future work.

2 Related Work

Our work builds upon the tremendous amount of previous research in program analysis and program transformations. In particular, we attempt to extend classical analyses and transformations to semantically higher level operations that are encapsulated in library functions. For example, one of our annotations specifies an abstract interpretation [9, 17] and another draws from the pointer analysis work of Wilson and Lam [28].

Compilers have long used hints and pragmas to guide optimizations such as register allocation and inlining, and to summarize procedure information such as whether a function has side effects. More recently, annotations have been used to guide dynamic compilation [13]. While annotations are not new, our use of them is new. First, our annotations describe function implementations, rather than call site-specific information. This means that application programs do not require annotations, so our annotations are hidden from the everyday user. Second, and more fundamentally, our advanced annotations can convey domain-specific information that other languages cannot. For example, annotators can define concepts, such as data distribution, that extend beyond those of the base language. However, unlike most hints and pragmas, the incorrect use of our annotations can lead to transformations that do not preserve the library's semantics.

Our work is closely related to partial evaluation [5, 6, 10], which improves performance by specializing routines for specific inputs. Partial evaluation combines inlining, constant propagation and constant folding to evaluate as much of the program as possible at compile time. Recent work in program specialization has generalized partial evaluation to the notion of staged optimizations, which can take place at compile time, link time or runtime [13, 14], and which can be applied to class libraries in object oriented programs [27]. All of these approaches specialize based on values of variables that are constant for some duration of the program. By contrast, our approach can specialize based on other criteria: For example, specialization can occur at a particular program point when the program moves into a particular program state. Our approach also can perform optimizations such as loop-invariant code motion that cannot be expressed using partial evaluation.

Software generators [23, 24] and program transformation systems [22] are compilers for domain-specific programming languages. While these systems provide so-

phisticated transformations of high level language constructs, they typically manipulate programs only at the syntactic level. Semantic properties, such as those resulting from dataflow analysis, are either awkward to express or completely unavailable. Our approach instead focuses on the exploitation of semantic, rather than syntactic, information.

There has been considerable work in formal semantics and formal specifications. In particular, Vandevoorde uses powerful analysis and inference capabilities to specialize procedure implementations [26]. However, complete axiomatic theories are difficult to write and do not exist for many domains. In addition, this approach depends on theorem provers, which are computationally intensive and only partially automated. Our work differs from these primarily in the scope and completeness of our annotations, which describe only specific implementation properties instead of complete behaviors.

Open and extensible compilers give the programmer complete access to the internal representation of the program [16, 12]. While these systems are quite general, they impose a considerable burden. To use them, the programmer needs to understand (1) general compiler implementation techniques, (2) how to configure the specific compiler they are using, and (3) how to express and execute their optimizations. Similarly, meta-object protocols provide sophisticated mechanisms for modifying the compilation of object oriented programs [8, 19], but they can be difficult to use. Our compiler limits configurability to a small but powerful set of capabilities, and provides a simple way to access them.

Finally, we note that our system is an instance of aspect-oriented programming [18]. In our case, the cross-cutting aspect is performance improvement, and our annotation language and compiler are specific mechanisms for implementing this aspect. An important feature of aspects is that they be separated from the rest of the code, and in our case this is achieved by placing the annotations in a separate file.

3 Annotation Language

The goal of the annotation language is to convey library-specific information to the compiler in a simple declarative manner. While it's clear that more sophisticated specifications could support more sophisticated optimizations, our goal is to show that a few simple annotations can enable many useful optimizations. Simplicity

is important because we expect our language users to be library experts who do not necessarily have expertise in compilers or formal specifications.

In designing the language, we studied several libraries to determine the most useful ways of optimizing them. We noticed that library operations could easily be integrated into many traditional optimizations, such as dead-code elimination, copy propagation and loop-invariant code motion. These optimizations are effective and well understood, and they require only minimal information to enable. For example, to enable loop invariant code motion, the annotations need to indicate which library procedures have no side-effects. We also observed that many library-specific optimizations replace a general-purpose library call with a more specific one that takes advantage of information about the calling context. This form of specialization not only improves performance, it often creates additional opportunities for traditional optimizations. Thus, our annotation language consists of two classes of annotations: *basic annotations* for enabling traditional optimizations, and *advanced annotations* for specifying library-specific specialization.

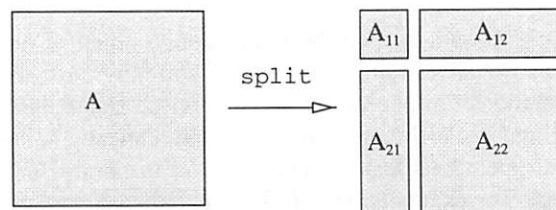
We present the annotation language by first describing the target library: the PLAPACK parallel linear algebra library [25]. The remainder of the section then describes the language constructs in detail, using a fragment of the PLAPACK annotations as a source of examples. These annotations capture the information used to produce the results in Section 5. A complete grammar is presented in the appendix.

3.1 The PLAPACK library

PLAPACK is a production-quality library for coding parallel linear algebra algorithms in C. It consists of approximately 40,000 lines of C code and provides parallel versions of the same kernel routines found in the BLAS [11] and LAPACK [2]. At the highest level, it provides an interface that hides much of the parallelism from the programmer.

A PLAPACK application operates on linear algebra objects, such as matrices and vectors, that are partitioned and distributed over the processors of the target computer. The application manipulates these objects indirectly through handles called *views*. A view specifies an index range that selects some or all of a distributed object for subsequent computations. PLAPACK contains routines to create new views, shift views, and split views into pieces. The following figure shows a four-way split

that logically divides a matrix into four smaller ones:



A typical algorithm starts with an entire object, like A , and splits it into manageable pieces. It computes directly on A_{11} , A_{12} and A_{21} , and then continues recursively by splitting the large remaining piece, A_{22} , until the entire data set has been visited.

Often, a view captures part of a matrix or vector that has special properties. Understanding and exploiting these properties can lead to significant performance improvements. For example, a view can select a region that resides entirely on one processor. Any computations on the data within this *local* view can be performed locally, without involving other processors. In the figure above, the four-way split yields one local view (A_{11}), one *column panel* (A_{21}), which resides on a column of processors, one *row panel* (A_{12}), which resides on a row of processors, and a large fully-distributed matrix (A_{22}). A view might also specify a region that is in tridiagonal form, allowing the use of specialized compute functions.

Our goal is to identify the library-specific properties that are relevant to optimization, and track them through the application program. For example, if an application splits a local view into two pieces, we can infer that the two new views are also local. The result of this analysis describes how the application manipulates objects with respect to library-specific properties such as distribution or data content. We can use this information to customize the library, or to select library routines that are better suited to the application.

Figure 2 shows a fragment of the annotations for PLAPACK. It specifies the two properties described above (distribution and data content) and gives the semantics of three PLAPACK routines: `PLA_Matrix_create`, which creates a new matrix, `PLA_Obj_vert_split_2` which splits a view into left and right pieces, and `PLA_Gemm`, which multiplies matrices.¹

¹In this figure, the `PLA_Gemm` interface has been simplified in insignificant ways to clarify the presentation. The actual routine accepts seven arguments instead of three.

```

(1)  %{
    #include "PLA.h"
    %}

    // --- Special case matrix distributions
    property Distribution = { General = none, ColPanel, RowPanel,
                                Local, Empty };
(2)  // --- Special case properties of the data
    property Contents = { Dense = none, Zero, Identity, Upper, Lower };

    // --- Procedure: Create a new distributed matrix

    procedure PLA_Matrix_create ( datatype, length, width, template,
                                align_row, align_col, new_matrix)
    {
(3)  on_exit { new_matrix --> view_1,
              DATA of view_1 --> data_1 }
    access { datatype, length, width, template, align_row, align_col }
    modify { new_matrix }
    analyze Distribution { view_1 = General; }
    }

    // --- Procedure: Split a matrix logically into two pieces

    procedure PLA_Obj_vert_split_2( obj, length, left, right)
    {
(4)  on_entry { obj --> view_1, DATA of view_1 --> data_1 }
    on_exit { left --> view_L, DATA of view_L --> data_1,
             right --> view_R, DATA of view_R --> data_1 }
    access { view_1, length }
    analyze Distribution {
(5)  (view_1 == General) => view_L = ColPanel, view_R = General;
      (view_1 == ColPanel) => view_L = ColPanel, view_R = Empty;
      (view_1 == RowPanel) => view_L = Local, view_R = RowPanel;
    }
    specialize {
      (view_1 Distribution == ColPanel) => replace "PLA_Copy_view(obj, view_L)";
    }
    }

    // --- Procedure: Compute C <- A * B

    procedure PLA_Gemm( A, B, C)
    {
    on_entry { A --> view_A, DATA of view_A --> data_A,
              B --> view_B, DATA of view_B --> data_B,
              C --> view_C, DATA of view_C --> data_C }
    access { data_A, data_B }
    modify { data_C }
    analyze Contents {
      ((data_A == Upper) && (data_B == Upper)) => data_C == Upper;
      ((data_A == Zero) || (data_B == Zero)) => data_C == Zero;
    }
    specialize {
      ((view_A Distribution == Empty) ||
       (view_B Distribution == Empty)) => remove;
      ((view_A Distribution == Local) &&
       (view_B Distribution == Local)) => replace "PLA_Local_gemm( A, B, C)";
      (view_A Contents == Upper) => replace "PLA_Trmm( A, B, C)";
    }
    }

```

Figure 2: Part of the annotations for the PLAPACK parallel linear algebra library. (1) The header provides access to definitions in the library header files. (2) The property annotations define abstract object states which are used for analysis and specialization. (3) Each library procedure has its own set of annotations. (4) The basic annotations summarize the dataflow and pointer behavior of the procedure. (5) The advanced annotations specify analysis rules for abstract interpretation and specialization rules that use the resulting information.

3.2 Basic annotations

Each library procedure can have a set of basic annotations that provides the information needed to support the Broadway compiler's dataflow analysis framework. This information allows the compiler to properly interpret library calls, and to integrate them into traditional optimization passes such as code motion, copy propagation and redundancy elimination.

A library procedure has access to many different data objects in the application program, including the arguments passed into it, and possibly global objects as well. In addition, many libraries create and manage complex pointer-based data-structures that are built up from many objects. We have found that in order to correctly analyze library calls, it is essential to accurately model these data-structures. Thus, the basic annotations provide two kinds of information: (1) a list of the objects that are accessible to the procedure and describe their structure, and (2) a list of those objects whose contents are accessed or modified by the procedure (the "uses" and "defs").

The information is specified using a technique similar to interval analysis [20]. Interval analysis concisely summarizes the effects of a procedure, so that the compiler can analyze any code that calls the procedure without re-analyzing the procedure itself. Our language allows the library annotator to explicitly summarize the dataflow and pointer effects for each library procedure [28]. In some cases, a modern compiler could derive this information automatically from the library source. However, there are conditions under which this is infeasible or impossible. Many libraries encapsulate functionality for which no source code is available, such as low-level I/O or communication routines. Even if source is available, it may be simpler to provide the information declaratively, especially if it is well known.

3.2.1 `on_entry` and `on_exit`

The `on_entry` and `on_exit` annotations specify the effects of a library procedure on objects that are organized into data structures. We model data structures by adding edges between the objects. The edges are directed and can be roughly interpreted as "points to". Each identifier in these annotations is either an input to the procedure (a formal parameter), or gives a name to an object that is reachable by following edges from an input. Like the formal parameters, each name is arbitrary

and is bound to actual objects at each procedure call site. The behavior of the procedure is summarized by showing the configuration before and after execution.

The `-->` operator indicates that the operand on the left points to the operand on the right. For example, in PLAPACK, each matrix parameter is passed as a pointer to a view structure, which in turn points to the underlying data. We can label an edge by providing an additional identifier followed by the `eof` keyword. In the example, we label each edge from a view to its data with the label `DATA`. This distinguishes it from any other things that a view might point to. Figure 3 depicts the pointer structure given by the annotations labeled (4) in Figure 2.

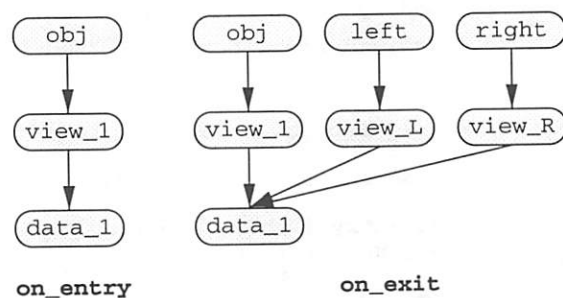


Figure 3: The effect of split on PLAPACK data structures.

We can use the keyword `null` on the right side to indicate the removal of an edge.

The data structures described in these annotations need not correspond exactly to the underlying implementation. In fact, it is often useful to make explicit some of the relationships that are only represented implicitly in the implementation. Many libraries contain objects that behave logically like pointers, such as handles, references and descriptors. We can use `on_entry` and `on_exit` to model all of these structures.

In addition to establishing new data structures, the `on_exit` annotation can declare that an object is a copy of another object, using the `copyof` keyword. We can exploit this information to perform high-level copy propagation on library objects.

3.2.2 `access` and `modify`

The `access` and `modify` annotations list the objects that are accessed or modified by the library procedure. The lists may contain formal parameters from the procedure input list, or object names introduced by the `on_entry` and `on_exit` annotations.

3.2.3 global

The `global` annotation declares global variables that can be analyzed along with the procedure parameters. These annotations simply provide a list of names that can be used to track global state information, and are not associated with a specific procedure. Like the `on_entry` and `on_exit` annotations, they need not correspond to actual global variables in the implementation. It is often useful to define global variables that model system states not explicitly represented by variables in the program. As examples, a global variable annotation can be used to track whether a library is properly initialized, or to maintain a record of outstanding asynchronous operations.

3.3 Advanced annotations

The advanced annotations define library-specific analyses and optimizations. The annotations are used to define a dataflow analysis problem consisting of a set of abstract object states and the effects of each library procedure on those states. The abstract states form a dataflow lattice and the library procedure effects serve as dataflow transfer functions. The analyzer propagates this information through the program to derive the abstract states of the actual program variables. A separate set of annotations uses this information to trigger library procedure specializations. Each specialization tests the abstract states of its input parameters to determine if the library call can be replaced by code that takes advantage of the context.

3.3.1 property

Each `property` annotation defines an abstract interpretation over objects in the program. The set of abstract values given in the curly braces form a two-level dataflow lattice. Figure 4 shows the lattice specified by the `Distribution` property given in Figure 2.

Because the lattices are only two levels high, whenever two program paths disagree on the state of an object the resulting *meet* results in lattice value \perp . We are considering ways to allow more complex lattices, such as multiple level lattices or infinite lattices, while still ensuring convergence. The keyword `none` allows a symbolic name to be assigned to the value \perp .

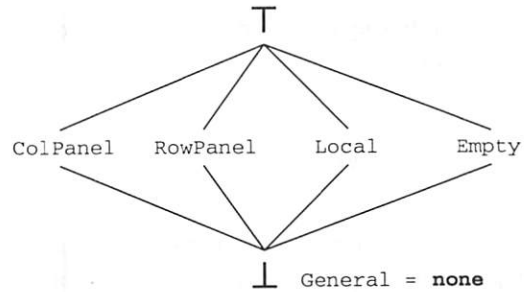


Figure 4: Lattice defined by the `Distribution` property.

3.3.2 analyze

Each library procedure can have a set of `analyze` annotations that describe how that procedure affects the properties of the objects it manipulates. Collectively, these annotations compose the dataflow transfer function for each abstract interpretation. Each statement in this annotation behaves as a logical implication: if the conditions on the left of the `=>` operator are true, then we conclude that the facts on the right are true. Each term in the condition is limited to testing the current property value of an object, or comparing to a constant. Each condition is a logic expression made up of these terms. In the absence of the `=>` operator, the facts are assumed without condition.

The PLAPACK annotations in Figure 2 show several examples of the `analyze` annotation. The part labeled (5) describes the effect of a vertical split on the distribution of various input view types. The matrix multiply procedure, `PLA_Gemm`, analyzes the contents of the matrices involved. For example, it expresses the fact that multiplying two upper-triangular matrices yields an upper-triangular matrix as a result.

When more than one analysis statement applies, the most specific one is chosen: the statement with the greatest number of conditions that are true, minus any that are false. For example, given an `analyze` annotation of the follow form:

```

analyze Foo {
    (A)          => C1;
    (A && B)     => C2;
    (A || B)     => C3;
}
  
```

If only `A` is true, then we would conclude `C1`. If both `A` and `B` are true, then we choose either `C2` or `C3`. Ties are

broken by preferring the statement that occurs earlier in the annotation.

3.3.3 specialize

Each procedure can specify a set of specializations that is triggered by the properties assigned to the input objects. The specializations modify the call site in the application code. Like the `analyze` annotations, each specialization is guarded by a condition, but these conditions are evaluated after abstract interpretation is complete. Unlike the `analyze` annotations, these conditions can refer to any combination of properties, and thus must provide the specific property name. The right side of the `=>` specifies either a literal code replacement, indicated by the `replace` keyword, or that the library call should simply be removed as indicated by the `remove` keyword.

The PLAPACK annotations in Figure 2 show three specializations for the matrix multiply procedure. The first causes the call to be removed whenever either of the inputs A or B refers to empty views. The second replaces the parallel matrix multiply routine with a local version if both A and B refer to local views. Finally, if the data indexed by A is upper-triangular, we can replace the general matrix multiply call with a call to a special triangular form that requires half the number of floating point operations.

4 The Broadway Compiler

This section describes the compiler's overall optimization strategy. The compiler consists mostly of traditional analysis and optimization algorithms, extended to use information from our annotation language. The individual transformations are straightforward and are not discussed. During a particular pass, the compiler refers to the annotations to find the information needed. Figure 5 shows the internal structure of the compiler and how the annotations are incorporated. We use a particular ordering of the passes that provides the most information for specialization, and then cleans up the customized code using traditional optimizations.

Pointer analysis. The first phase of the compiler performs pointer analysis. It not only tracks pointers in the application code, but also uses the `on_entry` and `on_exit` annotations to determine the data

structures manipulated by the library calls. Our pointer analysis algorithm builds a flow-sensitive "points-to" graph using the strategy described by Chase, et al [7].

Abstract interpretation. The second phase solves the analysis problems specified by the property annotations. The analysis framework assigns an abstract state to each object in the program and uses the `analyze` annotations to propagate this information through the program.

Enabling transformations. Dataflow analysis often loses interesting information because it acts conservatively with respect to control flow. For example, if a library procedure is used in two different ways, the analyzer will attempt to unify the information from both contexts. Thus, in the third phase the compiler uses any loss of information as a heuristic to drive enabling transformations, such as procedure integration, procedure cloning, loop peeling and node splitting. Since the properties are used to trigger specializations, using them to trigger these transformations is likely to enable many more specializations.

Specialization. In the fourth phase, the compiler uses the results of analysis along with `specialize` annotations to perform code customization. At each call site, the compiler looks for a specialization that matches the state of the variables. If a match is found, the call site is replaced. We have found that after specialization, it is often beneficial to repeat the abstract interpretation phase because the program modifications reveal new opportunities for optimization.

Traditional optimizations. Specialization often enables many opportunities for traditional optimizations. When a general library call is replaced by a special-case call, any arguments that are no longer used become candidates for dead-code elimination. Similarly, inlining a library procedure often reveals redundant computations and unnecessary copies of objects. Thus, in the final phase, we iterate over a small group of traditional optimization passes until no more improvements can be made.

The traditional optimization passes are extended to include library procedures. The basic annotations make this possible by providing the necessary information. During copy propagation, the `copyof` terms tell the compiler when copies of objects are created, and the `modify` annotation tells the compiler when those copies become invalid. Similarly,

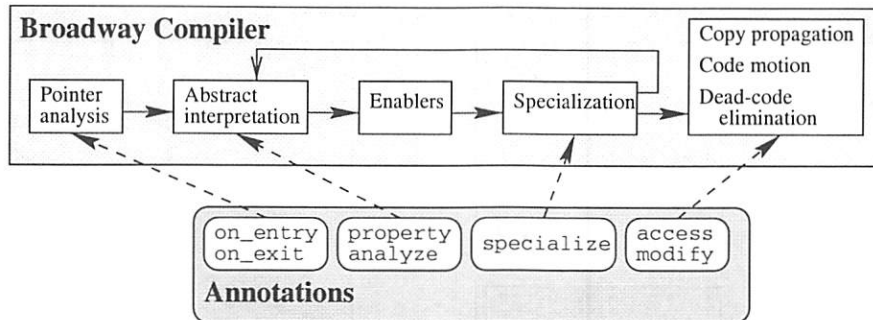


Figure 5: Annotations are incorporated into each phase of the compilation process.

the basic annotations indicate the lifetimes of the objects, allowing the dead-code elimination pass to properly identify dead library calls.

5 Results with PLAPACK

This section describes our experiences in applying our system to portions of two PLAPACK applications, a Cholesky factorization program and a code for solving Lyapunov equations [4].

For these experiments, our compiler performs all analysis automatically. Except for inlining, we perform the transformations manually according to the strategy described in Section 4. While our compiler is not yet complete, the individual transformations are all well-understood. Since the analysis and the overall compilation strategy are the enabling technologies behind these results, our manual transformations should not affect the results. The PLAPACK annotations were written by a person who is not a member of the PLAPACK implementation team. For purposes of comparison, the baseline programs were supplied by the PLAPACK group and written using the cleanest PLAPACK interface. The hand-optimized programs were written by PLAPACK experts. All results were obtained on a 40 node Cray T3E.

To gather these results we annotated 29 of PLAPACK's 113 externally visible routines, yielding an annotation file that was 323 lines. Our Broadway-optimized results focused on customizing one PLAPACK routine, the `PLA.Trsm()` routine, which is common to both the Cholesky and Lyapunov applications. The hand-optimized Lyapunov program did not limit itself to this same scope. Details concerning the hand-optimized version of the Cholesky program can be found in the litera-

ture [3].

Our annotations mimicked the hand optimizations by defining an abstract interpretation for describing the distribution of PLAPACK objects, leading to optimizations like those described in Section 3.1. (Unlike the example in Figure 2, we did not define the `Contents` property.) The basic idea is that while most PLAPACK procedures are designed to accept any type of view, the actual parameters often have special distributions. When this information is propagated into the procedure, it yields a variety of specialization opportunities. Uncovering these opportunities requires the compiler to analyze multiple layers of nested procedure calls. It is the encapsulation of these layered routines that makes the unoptimized routines both general and inefficient.

5.1 Performance Evaluation

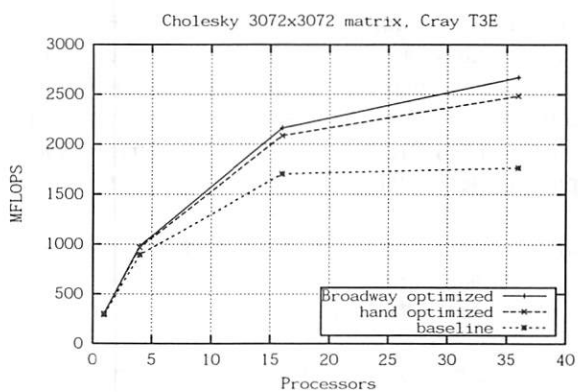


Figure 8: Scalability of the Cholesky programs as the number of processors grows.

Figure 6 shows the performance improvement of the Cholesky and Lyapunov programs. For fairly large matrices (6144×6144), the Broadway-optimized Cholesky

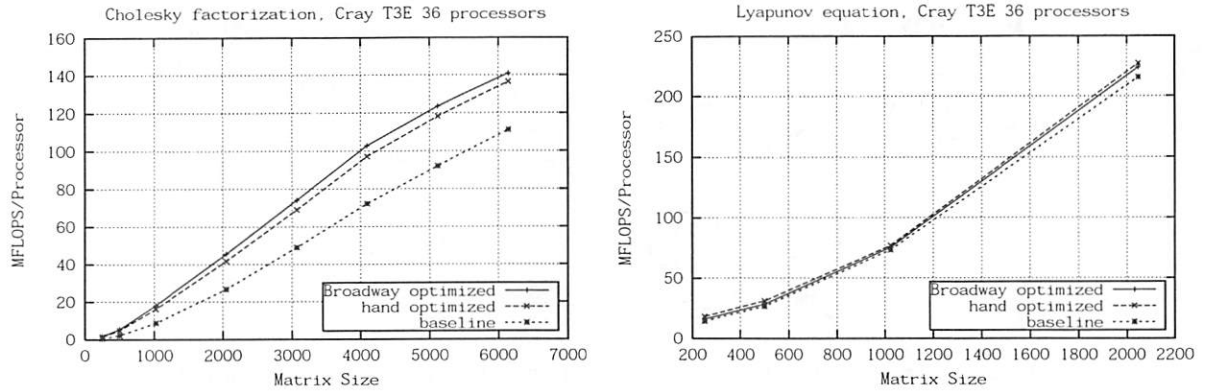


Figure 6: Performance comparison of hand-optimized and Broadway-optimized PLAPACK applications.

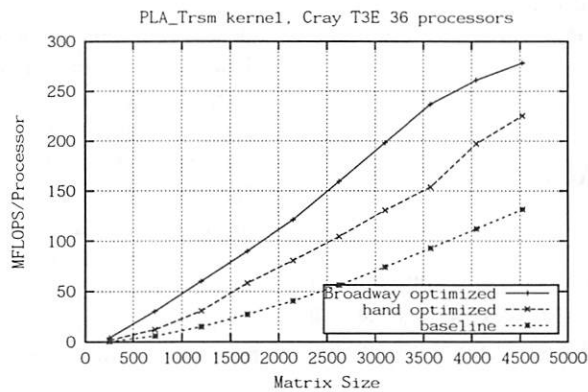


Figure 7: Performance comparison of hand-customized and Broadway-customized `PLA.Trsm()` function for the Cholesky program. For the Lyapunov program, the hand-customized `PLA.Trsm()` function matched the performance of the Broadway-customized version.

program is 26% faster than the baseline and the hand-optimized program is 22% faster than the baseline. For the Lyapunov program, the Broadway system does not perform as well as the manual approach, improving performance by 9.5% compared to the hand-optimized improvement of 21.5% for 250×250 matrices, and improving performance by 5.8% compared to 6.1% for 2000×2000 matrices. *The two approaches obtain identical performance on the `PLA.Trsm()` kernel, but the hand-optimized program performs a few additional optimizations to other parts of the code.*

Note that there is considerable room for further improving the Lyapunov program, since `PLA.Trsm()` only accounts for 11.6% of the execution time for 250×250 matrices, and only 5.8% of the time for 2000×2000 matrices. When our compiler is complete, we will apply our optimizations to all parts of the PLAPACK library, including the `PLA.Gemm()` routine, where Lyapunov spends a majority of its time.

Since our experiment focuses on the benefits of specializing the `PLA.Trsm()` routine, Figure 7 shows the performance difference between the generic `PLA.Trsm()` routine and the version that was customized for Cholesky by our compiler. Notice that we observe similar results for different numbers of processors. Figure 8 shows how the performance of the various Cholesky programs scale with the number of processors.

The results reveal several interesting points.

- A small effort yields a large benefit because the annotations only contain library knowledge, while all compilation expertise resides in the Broadway Compiler. The library annotator supplies the small but critical bits of information—such as specifying the conditions required to substitute a specific PLAPACK routine in place of a more general one—while the compiler analyzes the program, identifies opportunities for transformations, and manages a

number of optimization passes. This separation of concerns is beneficial because the performance improvements shown in Figure 7 come from the repeated application of a small number of transformations.

- Automation is desirable. Both the Cholesky and Lyapunov programs specialize the same PLAPACK routine, but they do so in slightly different ways because they invoke it in different contexts.
- An automated approach can apply all optimizations uniformly. There is no fundamental reason why the hand-optimized Cholesky factorization is not as efficient as ours, but the manual approach, which is quite invasive, did not employ one transformation that it could have.
- The effect of customization is more important for small matrices. For example, for a 1024×1024 matrix, the Broadway-optimized Cholesky factorization is 2.95 times faster than the base, and the hand-optimized is 2.47 times faster than the base. When matrices are small the improvements are larger because there is more overhead relative to matrix operations. Because dense linear algebra problems do not typically involve huge matrices, the small matrix cases is important for scaling to larger numbers of processors, and for supporting sparse matrix operations.

Closer examination of the Cholesky results reveal that specialization and dead code elimination account for almost all of the performance benefits, while high level copy propagation (where the copy operations are library routines) contributes insignificantly.

5.2 Language Evaluation

Simplicity. Our annotation language is small and simple. There are 15 keywords and a small number of simple concepts. The basic annotations require a knowledge of C and the library's data structures. The advanced annotations require a deeper knowledge of the library's implementation. Anecdotal evidence suggests that the language is intuitive. When shown the advanced annotations for PLAPACK, the head of the PLAPACK project claimed that they seemed "very natural."

While our language is quite simple, we believe that we can simplify the *use* of the language. Eventually, we imagine that basic annotations will only be specified

where static analysis fails. For example, a static analysis tool could guide the annotation by identifying routines that must be manually annotated.

Separation of Concerns. Our annotation language clearly separates the optimization information from the basic algorithm. By contrast manual optimization directly modifies the application source code, which complicates subsequent modification, reuse and maintenance. Moreover, we attempt to separate domain-specific information, which we place in the annotations, from compilation-specific information, which is embedded in the compiler. This separation of concerns simplifies both the library implementation and the specification of the annotations.

Generality. Our experiments show that our annotation language is effective when applied to PLAPACK. We believe that the language will also be effective for other libraries because the information conveyed by the basic annotations is fundamental to the analysis of any software, and the advanced annotations support abstract interpretation [9, 17], which is useful for modeling domain-specific information. In particular, such analysis is useful to any library that provides specialized routines that are tailored for specific contexts. For example, the Open GL graphics standard [21] can customize various matrix transformations to exploit particular properties of matrices and matrix operations. In operating systems, specialized file system I/O routines can be produced that are optimized for specific system states [10]: a specialized read routine can be created for the common situation in which the file is known to be open and the file position is correctly positioned to the next unread byte. As a final example, most layered systems can benefit from passing state information across layers [1], providing contextual information that can trigger the use of specialized routines.

Expressiveness. Because we have traded off generality for simplicity, our language is limited in the types of abstract interpretation that are supported. For example, our *property* annotations only allow enumerated lists of values, which correspond to finite lattices. In addition, our lattices have a fixed height of two. These restrictions ensure that our dataflow framework will converge, at the same time hiding the lattice-theoretic foundation of dataflow analysis from the annotator. We anticipate supporting more complex lattices, including integer ranges and restricted classes of infinite lattices. We

will enforce termination by putting bounds on the number of iterations of our dataflow analysis.

Our language is also restricted in the sense that there is no way to create dependences between different abstract interpretations.

6 Conclusions and Future Work

We have introduced a system that allows libraries to be both general and efficient. Applications can use a library's most general interface, and our compiler can customize the library implementation for different application needs. The key to our solution is an annotation language that conveys domain-specific information to the Broadway Compiler. The cost is that of annotating libraries, but the benefits are many: (1) Our compiler can perform domain-specific optimizations that are not possible without annotations; (2) our approach supports the use of cleaner, simpler interfaces, which leads to application code that is easier to maintain; (3) our approach provides a clear separation of concerns, as optimization information is encapsulated in the annotations rather than embedded in the application source code. In effect, the annotation language allows our compiler to treat libraries as semantically-rich but syntactically poor languages.

We have tested our technique by applying it to two programs written using the PLAPACK library. Our experience shows that (1) pointer-based C code can be analyzed with the help of our annotations, (2) our technique can produce significant performance improvements, even for a library that has already been carefully designed to achieve good performance, (3) a small number of simple annotations can be effective, and (4) the same set of annotations can be used to optimize multiple applications.

This work can be extended in many directions. When our compiler implementation is complete we will apply our transformations uniformly to a wider body of PLAPACK routines and a larger number of PLAPACK applications. We also plan to annotate other libraries, such as the standard math library, the MPICH [15] implementation of the Message Passing Interface, and perhaps Open GL [21]. More fundamentally, we are developing compilation strategies that allow us to optimize across multiple layers of libraries, and we are also exploring ways to extend our annotation language to support machine-specific customization.

Acknowledgments. We thank Robert van de Geijn for many insightful discussions about PLAPACK. We thank E Christopher Lewis and Yannis Smaragdakis for valuable comments on preliminary versions of this paper.

References

- [1] Mark B. Abbott and Larry L. Peterson. Increasing network throughput by integrating protocol layers. *IEEE/ACM Transactions on Networking*, 1(5):600–610, October 1993.
- [2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, second edition, 1995.
- [3] G. Baker, J. Gunnels, G. Morrow, B. Riviere, and R. van de Geijn. PLAPACK: high performance through high level abstractions. In *Proceedings of the International Conference on Parallel Processing*, 1998.
- [4] P. Benner and E.S. Quintana-Orti. Parallel distributed solvers for large stable generalized Lyapunov equations. In *Parallel Processing Letters*, 1998 (to appear).
- [5] A. Berlin. Partial evaluation applied to numerical computation. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, 1990.
- [6] A. Berlin and D. Weise. Compiling scientific programs using partial evaluation. *IEEE Computer*, 23(12):23–37, December 1990.
- [7] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. *ACM SIGPLAN Notices*, 25(6):296–310, June 1990.
- [8] S. Chiba. A metaobject protocol for C++. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, pages 285–299, October 1995.
- [9] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.
- [10] Crispin Cowan, Tito Autrey, Charles Krasice, Calton Pu, and Jonathan Walpole. Fast concurrent dynamic linking for an adaptive operating system. In *Proceedings of the International Conference on Configurable Distributed Systems*, May 1996.

- [11] J.J. Dongarra, I. Duff, J. DuCroz, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–28, 1990.
- [12] Dawson R. Engler. Incorporating application semantics and control into compilation. In *Proceedings of the Conference on Domain-Specific Languages (DSL-97)*, pages 103–118, Berkeley, October 15–17 1997. USENIX Association.
- [13] B. Grant, M. Mock, M. Philipose, C. Chambers, and S.J. Eggers. DyC: An expressive annotation-directed dynamic compiler for c. *Theoretical Computer Science*, to appear.
- [14] B. Grant, M. Philipose, M. Mock, C. Chambers, and S.J. Eggers. An evaluation of staged run-time optimizations in DyC. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 223–233, 1999.
- [15] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996.
- [16] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, December 1996.
- [17] Neil D. Jones and Flemming Nielson. Abstract interpretation: a semantics-based tool for program analysis. In *Handbook of Logic in Computer Science*. Oxford University Press, 1994. 527–629.
- [18] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, June 1999. Finland, Springer-Verlag LNCS 1241.
- [19] John Lamping, Gregor Kiczales, Luis H. Rodriguez Jr., and Erik Ruf. An architecture for an open compiler. In *Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures*, 1992.
- [20] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufman, San Francisco, CA, 1997.
- [21] Jackie Neider, Tom Davies, and Mason Woo. *Open GL Programming Guide*. Addison-Wesley, 1996.
- [22] J. N. Neighbors. Draco: A Method for Engineering Reusable Software Systems. In T. J. Biggerstaff and C. Richter, editors, *Software Reusability*, volume I — Concepts and Models, chapter 12, pages 295–319. ACM press, 1989.
- [23] Y. Smaragdakis and D. Batory. Application generators. *Encyclopedia of Electrical and Electronics Engineering*, to appear.
- [24] Yannis Smaragdakis and Don Batory. DiS-TiL: a transformation library for data structures. In *USENIX Conference on Domain-Specific Languages (DSL-97)*, October 1997.
- [25] Robert van de Geijn. *Using LAPACK – Parallel Linear Algebra Package*. The MIT Press, 1997.
- [26] Mark T. Vandevoorde. *Exploiting Specifications to Improve Program Performance*. PhD thesis, MIT, Department of Electrical Engineering and Computer Science (also MIT/LCS/TR-598), 1994.
- [27] Eugen N. Volanschi, Charles Consel, and Crispin Cowan. Declarative specialization of object-oriented programs. *SIGPLAN Notices, Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-97)*, 39(1):286–300, October 1997.
- [28] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, La Jolla, California, 18–21 June 1995.

A Annotation language grammar

This appendix presents the complete grammar for the annotation language. We use the following type face conventions: *Italic font* for non-terminals, **bold typewriter font** for literal terminals including keywords, and SMALL CAPS for the lexicographic terminals such as identifiers and C code fragments. In addition, we use the square brackets to represent optional components, and the star to represent repetition of a component.

A.1 Overall format

```

Annotations → Header
              Annotation *

Header      → %{
              C-CODE
              %}

Annotation → Property_ann
            | Global_ann
            | Procedure

```

A.2 Globals and properties

```

Global_ann → global { Identifiers }

Property_ann → property { Properties }

Properties → Property [ , Properties ]

Property → IDENTIFIER [ = none ]

```

A.3 Procedures

```

Procedure → procedure IDENTIFIER ( identifiers )
           { Proc_ann * }

Proc_ann → Structure_ann
          | Def_use_ann
          | Analyze_ann
          | Specialize_ann

```

A.4 Object structure

```

Structure_ann → on_entry { Structures }
               | on_exit { Structures }

Structures → Structure [ , Structures ]

Structure → Source --> Target
           | IDENTIFIER copyof IDENTIFIER

Source → [ IDENTIFIER of ] IDENTIFIER

Target → IDENTIFIER
        | null

```

A.5 Definitions and uses

```

Def_use_ann → access { Identifiers }
             | modify { Identifiers }

Identifiers → IDENTIFIER [ , Identifiers ]

```

A.6 Analyze

```

Analyze_ann → analyze IDENTIFIER { Rule * }

Rule → [ Condition => ] Consequence ;

Condition → IDENTIFIER [ IDENTIFIER ] == IDENTIFIER
           | IDENTIFIER [ IDENTIFIER ] == CONSTANT
           | ( Condition )
           | Condition && Condition
           | Condition || Condition

Results → Result [ , Results ]

Result → IDENTIFIER = IDENTIFIER

```

A.7 Specialize

```

Specialize → specialize { Spec * }

Spec → Condition => Replacement ;

Replacement → remove
             | replace C-CODE

```

A Case for Source-Level Transformations in MATLAB

Vijay Menon

*Department of Computer Science
Cornell University
Ithaca, NY 14853*

vsm@cs.cornell.edu

<http://www.cs.cornell.edu/Info/People/vsm>

Keshav Pingali

*Department of Computer Science
Cornell University
Ithaca, NY 14853*

pingali@cs.cornell.edu

<http://www.cs.cornell.edu/Info/Projects/Bernoulli>

Abstract

In this paper, we discuss various performance overheads in MATLAB codes and propose different program transformation strategies to overcome them. In particular, we demonstrate that high-level source-to-source transformations of MATLAB programs are effective in obtaining substantial performance gains regardless of whether programs are interpreted or later compiled into C or FORTRAN. We argue that automating such transformations provides a promising area of future research.

LAB well-suited for prototyping code. (A compiler that translates MATLAB code to C is available from MathWorks.) Third, the *mex-file* facility makes it easy to invoke compiled C or FORTRAN functions from the MATLAB interpreter. Finally, a full set of numerical and graphics libraries for many applications domains like computational finance and signal processing is available.

In general, there are overheads in the interpretation of programs in high-level untyped languages like MATLAB that are not there in executing code compiled from more conventional general-purpose languages like C or FORTRAN. The most important of these overheads in MATLAB are the following.

1 Introduction

MATLAB is a programming language and development environment which is popular in many application domains like signal processing and computational finance that involve matrix computations. There are many reasons for its popularity. First, MATLAB is a relatively high-level, untyped language in which matrices are a built-in data type with a rich set of primitive operations. Second, MATLAB programs are interpreted, making MAT-

- *Type and shape checking/dispatch*: A variable can be introduced in MATLAB programs without declaring its type or shape. Therefore, execution of the MATLAB statement `C = A*B;` can require a computation as simple as the product of two doubles or as complicated as the product of two matrices *A* and *B* containing complex entries. The interpreter must check types and shapes of operands in expressions for compatibility, and dispatch to the right routine for carrying out the appropriate operation. In traditional compiled languages, by contrast, the types and shapes of matrices would be known to the compiler, so at runtime it

⁰This work was supported by NSF grants CCR-9720211, EIA-9726388 and ACI-9870687.

is only necessary to check that the number of columns of A is equal to the number of rows of B .

- *Dynamic resizing:* Since matrix sizes are not declared by the programmer, the MATLAB interpreter allocates storage for a matrix on demand. If x is a vector and an attempt is made to write to element $x(i)$ where i is outside the current bounds of the vector, MATLAB allocates new storage for a larger vector and copies over elements from the old vector into the new storage. In FORTRAN or C, it is the responsibility of the programmer to allocate a large enough matrix, and any attempt to write into a location outside the index space of the matrix is an error.
- *Array bounds checking:* The interpreter must check indexed accesses of array elements to ensure that the access is within array bounds. In conventional languages, static analysis can use array declarations to eliminate bounds checks in many cases.

Techniques for reducing such interpretive overheads may be useful not only for MATLAB programs but for programs in other domain-specific languages many of which are also high-level, untyped, interpreted languages.

One approach is to translate MATLAB programs into programs in a conventional language like C or FORTRAN, and attempt to eliminate these overheads when compiling the C/FORTRAN code down to machine code. Several projects in both academia and industry have taken this approach [2, 4, 5, 9, 13, 18, 19, 20]. The mex-file interface described earlier can be used to invoke compiled routines from the interpreter, permitting the programmer to use the familiar MATLAB execution environment when running compiled code. However, as MATLAB lacks variable declarations, generation of efficient C or FORTRAN requires inference of types, shapes, and sizes. Unfortunately, compiler techniques to automatically infer these properties without additional user input have had limited success, as we discuss later in this paper.

In this paper, we will make the case for a very different approach to reducing these overheads — by using source-level transformations of MATLAB code. The MATLAB community has developed a number of programmer tricks [12] to enhance per-

formance of MATLAB codes. Surprisingly, these ideas have never been studied in the context of compilation. The conventional compiler approach, described above, replaces matrix operations with loops and indexed array accesses which are then optimized using standard compiler technology. The source-level approach advocated in this paper has the opposite effect since it replaces loops and indexed accesses by high-level matrix operations! This is somewhat counter-intuitive because type and shape checks, dynamic resizing and array bounds checks are not explicit in high-level matrix programs, so it is not obvious how these overheads are reduced by source-level transformations. Nevertheless, we show that such source-level transformations are beneficial regardless of whether the transformed code is executed by the MATLAB interpreter or compiled to C or FORTRAN.

The rest of this paper is organized as follows. A more detailed discussion of the overheads of interpreting MATLAB programs is given in Section 2. Section 3 discusses how the MCC MATLAB to C compiler attempts to eliminate these overheads. We also show the effect of using various compiler flags in MCC such as the `-i` flag to eliminate array bounds checks. Section 4 describes the source-level transformations of interest to us and evaluates their effect on performance if the resulting code is interpreted by the MATLAB interpreter. Section 5 argues that source-level transformations are useful even if the transformed code is compiled to native code by the MCC compiler. Section 6 compares our work with previous work. Section 7 gives a sketch of how these transformations can be performed automatically by a restructuring compiler; details of an implementation can be found in a companion paper [14].

2 Interpretation Cost of MATLAB programs

Our work-load in this paper is the FALCON benchmark set from the University of Illinois, Urbana [3]. This is a set of 12 programs from the problem domain of computational science, and it contains iterative linear solvers (CG,QMR), finite-difference solvers for pdes (CN,SOR), preconditioner computation for iterative linear solvers (IC), etc. These benchmarks are described in Table 1. In terms of MATLAB behavior, De Rose [3] groups the programs into three separate categories. *Library-*

intensive programs (CG, Mei, QMR, SOR) operate upon entire matrices via high-level operations or routines. These codes contain few, if any, indexed accesses into arrays. *Elementary-operation-intensive* programs (CN, Di, FD, Ga, IC) operate upon elements of matrices via loops. Virtually the entire execution time is spent within loop nests operating on array elements. *Memory-intensive* programs (AQ, EC, RK) require considerable memory management overhead in the form of dynamic resizing.

To measure the overhead of MATLAB interpretation, we would have liked to execute our benchmark suite on a suitably instrumented MATLAB interpreter. Unfortunately, the MathWorks interpreter is proprietary code, so we did not have access to the source. We considered using publicly available MATLAB-like interpreters and even instrumented one of them (Octave [6]), but we found that they were sufficiently different from the MathWorks interpreter that we could not draw meaningful conclusions about the performance of the MathWorks interpreter from experiments on other interpreters. For example, Octave is written in C++ and it uses the type-dispatch mechanism of C++ to implement the type checking and type dispatch required to execute MATLAB matrix operations, so there is no direct way to measure this overhead. Therefore, we had to make do with measuring the effect of program transformations on overall performance. In this section, we describe the interpretive overheads in more detail.

2.1 Type and Shape Checking

Unlike in C or FORTRAN, array variables in MATLAB programs can be introduced without type declarations. Furthermore, a single variable can name matrices of different types, shapes and sizes in different parts of the program. The complexity that this introduces in the interpreter can be appreciated by considering the assignment $C = A*B$. The type and shape of A and B determine what computation is performed, so $A*B$ may refer to scalar-scalar multiplication, scalar-matrix multiplication, or matrix-matrix multiplication where neither, either, or both of the arguments are complex. Each possible combination specifies a different kind of computation. Furthermore, the interpreter may also test for special cases where, for example, one of the arguments is a row or column vector. While a vector could

be treated just as any other matrix, more efficient underlying implementations exist for multiplying a matrix with a vector. Finally, the interpreter may also have to test for legality; in the case of matrix-matrix multiplication, the second dimension of A and the first dimension of B must conform.

The MATLAB interpreter tests for all of the above possibilities each time it encounters the $*$ operator. However, examining the context in which the expression occurs may reveal that the tests are redundant or even completely unnecessary. Consider, for example, the $*$ operator in the code:

```
for i = 1:n
    y = y + a*x(i);
end
```

In this case, the cost of the checks is magnified in the interpreter as they are performed in each iteration of the loop. However, note that $x(i)$ must be scalar, so no test is needed for it. Without additional information, the type and shape of a must be checked, but since a is not modified within the loop, these properties need to be tested just once, before the loop is executed.

2.2 Dynamic Resizing

In C or FORTRAN, storage for an array is allocated before its elements are computed. Since there are no variable declarations in MATLAB, storage for matrices and vectors is allocated incrementally during program execution. An attempt to write into a matrix element outside the bounds of the matrix causes the system to reallocate storage for the entire matrix, copying over all elements from the old storage to the newly allocated space. In loops, such memory management overheads can become prohibitively expensive. Consider the following code:

```
for i = 1 : 10000
    x(i) = i;
end
```

If x is initially undefined, the interpreter “grows” the vector incrementally during loop execution. On a Sparc 20, the MATLAB interpreter requires 14.2 seconds to execute this loop. However, it is clear before the execution of the loop that x will grow to

Benchmark	Flops	Lines of Code
AQ Adaptive Quadrature Using Simpson's Rule	3.6×10^5	87
CG Conjugate Gradient method	3.7×10^7	36
CN Crank-Nicholson solution to the heat equation	2.2×10^6	29
Di Dirichlet solution to Laplace's equation	1.9×10^6	39
FD Finite Difference solution to the wave equation	2.3×10^6	28
Ga Galerkin method to solve the Poisson equation	1.3×10^6	48
IC Incomplete Cholesky Factorization	7.6×10^5	33
Mei Generation of 3D-surface	1.7×10^7	28
EC Two body problem using Euler-Cromer method	3.3×10^5	26
RK Two body problem using 4th order Runge-Kutta	4.4×10^5	66
QMR Quasi-Minimal Residual method	1.2×10^8	91
SOR Successive Over-relaxation method	8.9×10^7	29

Table 1: Falcon Benchmark Suite

10,000 elements. If a vector \mathbf{x} of this length is pre-allocated before the loop begins, the loop executes in 0.37 seconds. In other words, repeated reallocation in the loop slows the loop down by a factor of 40 in this case. Note that the MATLAB interpreter only allocates the minimal amount of memory each time. That is, if the vector is not preallocated before the loop, then, on each iteration, the vector is reallocated into a memory block one element larger.

Interestingly, this overhead in the MATLAB interpreter is not nearly as significant for two dimensional arrays. Consider:

```
for i = 1 : 100
  for j = 1 : 100
    y(i,j) = i;
  end
end
```

Again, we have an array eventually resized to hold 10,000 elements. However, in this case, reallocation is not done on each iteration. In the first iteration of the i loop, each iteration of the j loop resizes y by an additional column. In subsequent iterations of the i loop, only the first iteration of the j loop causes resizing. That iteration resizes y to an $i \times 100$ array. No other iteration triggers resizing. When y is initially undefined, the interpreter requires 0.67 seconds to execute the above loop. With y already allocated, the interpreter requires 0.48 seconds.

While it is clear that the two dimensional case requires fewer memory reallocations, it is also true that it requires less data copying. In the one dimensional case, iteration i requires copying of $i - 1$ data elements. An entire n^2 element vector (where

$n = 100$ in our example) requires $O(n^4)$ copies. On the other hand, for an equivalent size $n \times n$ array, $O(n^2)$ copies are required for the first row, and $(i - 1) * n$ copies are required for each subsequent row i . Thus, the total number copies for the two dimensional array is $O(n^3)$, asymptotically smaller than the one dimensional case.

We conclude that the overhead of dynamic resizing is most important when vectors are resized within loops.

2.3 Array Bounds Checking

Indexed accesses into arrays are another source of run-time checks in MATLAB. Consider the following code:

```
x(i) = y(i);
```

The index i is checked to see if it is within the bounds of \mathbf{x} and \mathbf{y} . If it is not within the bounds of \mathbf{x} , it triggers resizing as explained above. If it is not within the bounds of \mathbf{y} , an error is reported.

As with type and shape checks, array bounds checks are often redundant, as in the code:

```
for i = 2:n-1
  x(i) = x(i-1)+x(i+1);
end
```

As before, the loop magnifies the overhead in the interpreter since three checks are performed in each

iteration. Clearly, the three checks performed on the inner statement are redundant and can be collapsed to one. The array x must contain at least $i+1$ elements for the statement to be legally executed. The other two checks are subsumed by this check. Furthermore, this remaining check need not be performed each iteration. If x does not contain at least n elements, it is clear that the loop cannot execute correctly.

3 Conventional Compilation

In this section, we examine the standard approach to compiling away the interpretive overheads of MATLAB programs. The key idea is to translate MATLAB code into C or FORTRAN programs, making type checking, dynamic resizing, etc. explicit and therefore amenable to optimization. We describe how the commercial MathWorks MCC compiler removes these overheads and quantify its effectiveness.

3.1 Type Inference

The MCC compiler attempts to eliminate type and shape checks through the use of type and shape inference. The most sophisticated type inference algorithm in the literature is the one in the FALCON compiler of De Rose et al. [3, 4]. The algorithm used in MCC is unpublished, but it appears to be similar. The high level idea is to generate a system of type equations relating the types and shapes of different variables and solve this system to determine what these types and shapes can be. In particular, type and shape information of inputs to expressions can be used to determine type and shape information of outputs.

MCC operates at the level of single MATLAB functions, or *m-files*. When MCC performs inference on a function, the result is a forward propagation algorithm in which the types and shapes of parameters and initialized local variables in a program are used to determine the types of intermediate and output variables. This, however, requires that type and shape information be known for parameter variables; this is very difficult to infer automatically. MCC's strategy is to generate two versions of compiled code for each function: one that assumes that

all inputs are real and one that assumes all are complex. The compiler then inserts an initial run-time test at the beginning of execution to determine if, in fact, all inputs are real.

Figure 1a demonstrates the effectiveness of the MCC compiler on the Falcon benchmarks on a Sun Sparc 20 machine. For this set of measurements, no additional compile-time flags were used; we will consider the effect of the two available optimization flags below. Note that the compiler is most effective on loop-nest intensive codes. On codes that are not loop intensive and predominantly utilize high-level operations, such as CG, QMR and SOR, the compiler shows no performance benefit. Surprisingly, the compiled code actually shows a slight slowdown in two of these cases. In these cases, type checking and type dispatch overhead is relatively insignificant compared to the time taken by the actual computation.

Users may obtain better performance from MCC by directing it to perform additional optimizations through the use of compilation flags. These optimizations, unlike the default ones, are potentially unsafe since they may be illegal for some programs. One unsafe optimization, triggered by a `-r` option to the compiler, eliminates all tests for complex types and as a result, generates code that assume all computation is on real numbers. Obviously, this will not produce correct results for programs operating on complex numbers. In the Falcon benchmarks, however, this optimization is applicable on all but one of the benchmarks (Mei). Figure 1b illustrates the effect of this optimization on the remaining eleven benchmarks.

There are noticeable performance gains in only three benchmarks (Di, IC, RK). As mentioned above, the default behavior of MCC is to generate two versions of compiled code in which one version assumes that all parameters are real. In this version, the compiler is usually able to determine that all intermediate and output variables are real as well. If this is the case, the compiler will eliminate all checks for complex values. The only additional advantage brought by the `-r` option is to eliminate the initial test on input values which, as seen in Figure 1b, is negligible. However, real input variables may not necessarily imply real intermediate or output variables. Certain operations, such as a square root, may produce complex values from real ones. In the presence of such operations, type inference will fail to eliminate all type checks. For example, in the In-

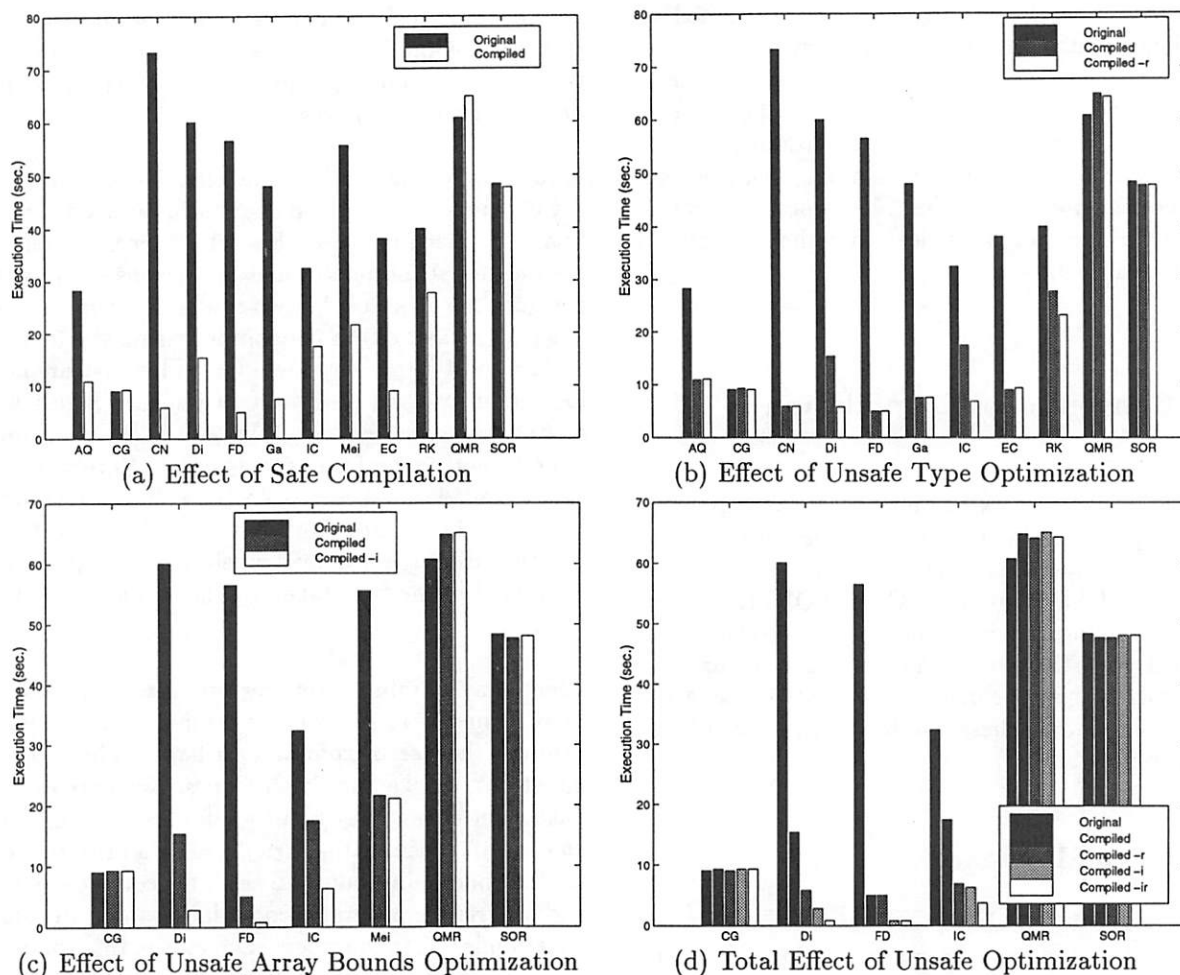


Figure 1: MCC Compilation

complete Cholesky benchmark, each column of the matrix is scaled by the square root of the diagonal element. Even though the input matrix is real, MCC is unable to infer that the result matrix will also be real. In these cases, the `-r` option, if applicable, can noticeably enhance performance.

3.2 Array Bounds Optimization

The MCC compiler does not optimize away any array bounds checks. However, the programmer can trigger bounds check elimination by using another compilation flag (the `-i` flag). As described earlier, this is not safe even for correct programs because out-of-bounds writes to an array are used to trigger dynamic resizing. Furthermore, if the code has an out-of-bounds read access, the compiled code generated by using this flag may produce either incorrect

results or catastrophic errors.

In the Falcon benchmarks, bounds check elimination is valid in seven of the twelve programs (in the remainder, the interpreter either halts with an error or produces incorrect results). The effect of using the `-i` compiler flag on these programs is shown in Figure 1c. There are substantial improvements in three of the seven benchmarks for which this flag is legal (DI, FD, IC). The remaining benchmarks predominantly utilize higher-level functions and contain few, if any, subscripted references to arrays.

3.3 Discussion

The overall effect of compilation, using compiler flags to eliminate type and shape checks as well as array out of bounds checks where legal, is shown in

Figure 1d. Eliminating type checks by using the `-r` flag is useful in the Di, IC and RK benchmarks while elimination of array bounds checks by using the `-i` flag is most effective in the Di, FD and IC benchmarks. Note that when a compiler flag is unsafe for a program, it may still be possible to apply the corresponding optimization to just a portion of the program. In the Galerkin benchmark, for example, dynamic resizing occurs within the function, and so the `-i` option will generate erroneous code. However, it does not occur within the innermost loop, where array bounds checks are most expensive. Performance measurements at this finer level of granularity require access to the MathWorks interpreter and compiler.

4 Source-Level Optimizations

In this section, we discuss source-level transformations of MATLAB programs and show how they can be used to reduce interpretive overhead. At first glance, this may seem to be a counterintuitive idea since the language provides no direct means of instructing an interpreter when and when not to perform various checks. *The key insight is that these overheads are most significant in loops, so loops can be transformed to eliminate interpretive overhead.* In this section, we discuss three different source-level transformations and show how they improve the efficiency of our work-load. In the next section, we compare these performance improvements with the performance improvements obtained by the MCC compiler.

4.1 Vectorization

Vectorization transforms loop programs into high-level matrix operations. This is similar to vectorization for vector supercomputers; in both cases, the key is to map a sequence of operations on array elements into one or more high-level operations on entire arrays. On vector hardware, these array operations can be executed more efficiently than loops with scalar operations. A similar gain in efficiency is possible in MATLAB interpretation. Loops slow down MATLAB programs by magnifying the overhead of statements contained within the loop. Any type checks or array bounds checks performed on a statement within the loop will be repeated for every

iteration even though multiple checks may be redundant. However, for higher level MATLAB operations that act on entire matrices and vectors, these checks are performed only once. Hence, the performance benefit of high level operations can be very large.

Consider the execution profile of the Galerkin benchmark in Figure 3a. This loop nest represents a small portion of the program but it is clearly the bottleneck in performance since 97% of the execution time of the entire program is spent within this loop nest. However, the entire loop nest can be vectorized by realizing that it is actually performing a vector-matrix-vector multiplication. When this loop nest is replaced by equivalent matrix-vector operations, the resulting profile is as shown in Figure 3b. This transformation enhances the performance of the loop nest by a factor of 250, and the performance of the entire benchmark is increased 100-fold!

Of course, vectorization is not always applicable. Many programs such as CG and QMR in the Falcon suite already extensively utilize higher-level operations and contain no for-loops. In other programs, such as the Dirichlet code in Figure 4, expensive for-loops exist but cannot be mapped to higher-level operations. In this case, the dependences due to `U` prevent either of the for-loops from being vectorized.

Vectorization is applicable to five of the twelve Falcon benchmarks. The effects on each of these five programs is shown in Figure 2a. In two programs (FD, Ga), the effects are dramatic since they result in more than 30-fold improvements. In these cases, vectorizable loops were responsible for nearly all the original execution time. In one case (Di), the effect is minor. Here, the vectorizable loop took only a minor portion of the original execution time.

4.2 Preallocation

As discussed in Section 2.2, resizing of arrays can result in significant memory management overhead due to repeated reallocation and copying. In many cases, the final size of the array can be easily inferred. When this is the case, it is often safe to preallocate the entire array at once. Consider the original code for the Euler-Cromer program in Figure 5a.

87% of program execution time is spent in the lines

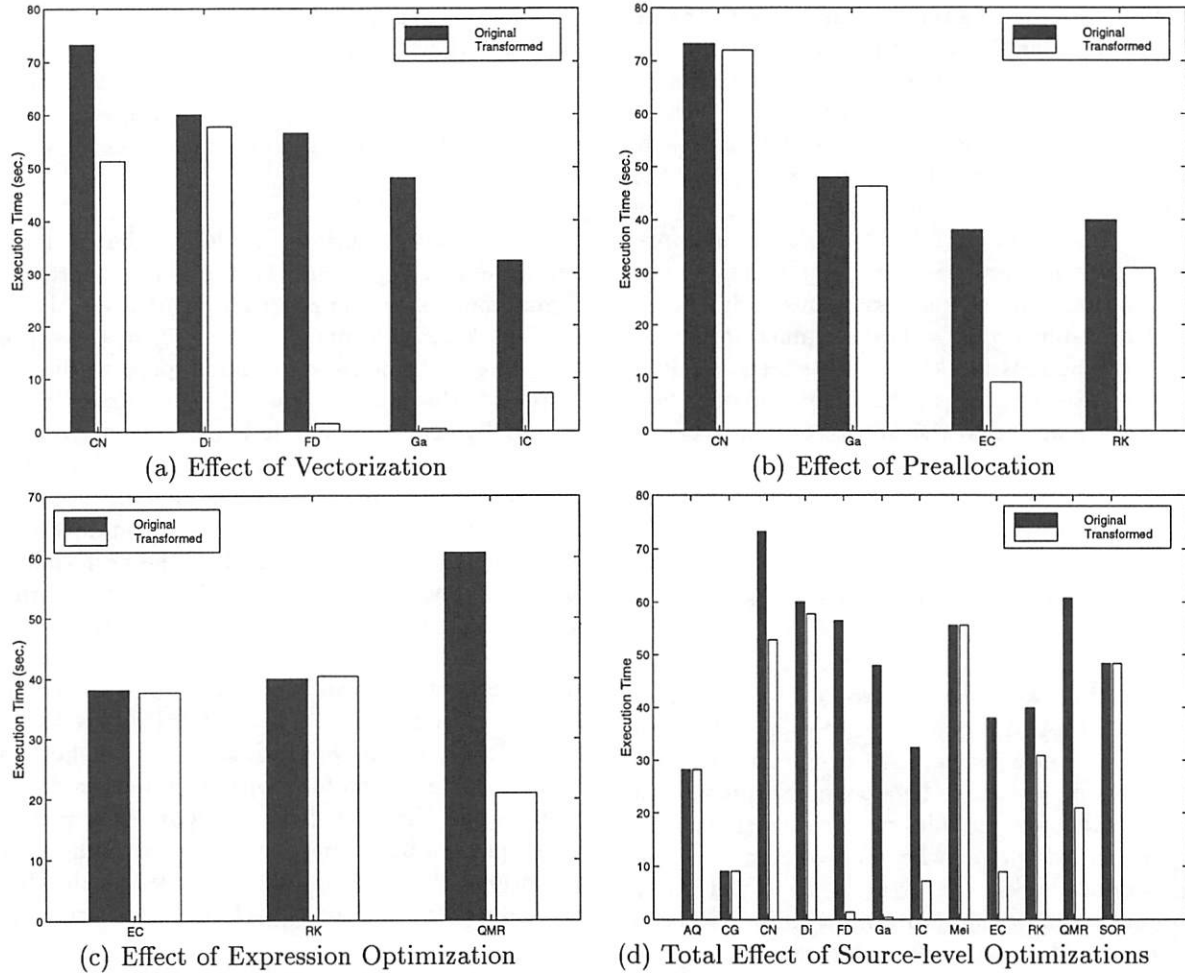


Figure 2: Source-to-Source Optimizations

shown. In this case, each of the arrays shown above is undefined prior to execution of this loop, so the array is resized on each iteration of the loop. However, it is clear in this case that ultimately, each array will be of length `nstep`. There is no way to declare the size of a matrix in MATLAB, but an indirect way to accomplish the same goal is to use the `zeros` operator that creates an array of a desired size and initializes its values to 0. Therefore, statements of the following form can be used to avoid resizing:

```
rplot = zeros(1,nstep);
```

When preallocation is done for all arrays, we obtain the profile shown in Figure 5b. Each of these statements is now significantly faster (by a factor of more than seven). As a result, the entire benchmark is faster by a factor of roughly four.

Unfortunately, simple preallocation as above cannot eliminate all instances of dynamic resizing. For example, in the AQ code, the final size of the array cannot be determined a priori. While more complex strategies such as preallocating an estimated size or explicitly growing the array in an exponential manner may reduce this cost, we have not attempted to do so in this paper.

Finally, even when preallocation can be done, the performance benefits will differ from case to case. The Euler-Cromer code represents a relatively extreme case since several one dimensional arrays are resized, and the final size of each (over 6,000) is fairly large. As mentioned in Section 2.2, the resizing overhead is significantly less with two dimensional arrays.

Dynamic array resizing occurs in five of the twelve Falcon benchmarks. In four of these cases (AQ is

a) Original Galerkin Code:

```

39: for i=1:N
0.24s, 1% 40:   xtemp = cos((i-1)*pi*x/L);
0.23s, 1% 41:   for j=1:N
16.36s, 88% 42:     phi(k) = phi(k) + a(i,j)*xtemp*cos((j-1)*pi*y/L);
1.26s, 7% 43:   end
0.03s, 0% 44: end

```

b) Transformed Galerkin Code:

```

0.01s, 1% 39: xtemp_se = cos((0:N-1)*pi*x/L);
0.06s, 9% 40: phi(k) = phi(k) + xtemp_se*a*cos((0:N-1)*pi*y/L)';

```

Figure 3: Effect of Vectorization on Galerkin Benchmark

Original Dirichlet Code:

```

53: for j=2:(m-1),
0.72s, 1% 54:   for i=2:(n-1),
33.91s, 53% 55:     relx = w*(U(i,j+1)+U(i,j-1)+U(i+1,j)+ U(i-1,j)-4*U(i,j));
20.39s, 32% 56:     U(i,j) = U(i,j) + relx;
6.13s, 10% 57:     if (err<=abs(relx))
0.06s, 0% 58:       err=abs(relx);
0.02s, 0% 59:     end
2.13s, 3% 60:   end
0.05s, 0% 61: end

```

Figure 4: Dirichlet Benchmark

a) Original Euler-Cromer Code:

```

18: for istep=1:nstep
8.75s, 18% 19:   rplot(istep) = norm(r);
8.05s, 17% 20:   thplot(istep) = atan2(r(2),r(1));
6.99s, 15% 21:   tplot(istep) = time;
8.89s, 19% 22:   kinetic(istep) = .5*mass*norm(v)^2;
8.49s, 18% 23:   potential(istep) = - GM*mass/norm(r);
...
0.15s, 0% 29: end

```

b) Transformed Euler-Cromer Code:

```

18: for istep=1:nstep
1.12s, 10% 19:   rplot(istep) = norm(r);
0.85s, 7% 20:   thplot(istep) = atan2(r(2),r(1));
0.24s, 2% 21:   tplot(istep) = time;
1.78s, 15% 22:   kinetic(istep) = .5*mass*norm(v)\^{}2;
1.44s, 13% 23:   potential(istep) = - GM*mass/norm(r);
...
0.12s, 1% 29: end

```

Figure 5: Effect of Preallocation on Euler-Cromer Benchmark

a) Original QMR Code:

```
19.10s, 70% 50: w_tld = ( A'*q ) - ( beta*w );
```

b) Transformed QMR Code:

```
0.80s, 9% 50: w_tld = ( q'*A )' - ( beta*w );
```

Figure 6: Effect of Expression Optimization on QMR Benchmark

the exception), the eventual size of resized arrays is easily determined. Figure 2b highlights the effect of preallocation on these four benchmarks.

4.3 Expression Optimization

Finally, we consider a source-level transformation not directly motivated by MATLAB overheads. Instead, we are motivated by the naivete of MATLAB's evaluation process. Unlike an optimizing compiler, the MATLAB interpreter does not consider the best manner in which to compute an expression. Instead, it blindly computes it in the most straightforward manner. Consider the profile information from the QMR benchmark in Figure 6a.

In an eighty line program, this single statement requires 70% of the entire execution time. Closer examination of this program reveals that `beta` is a scalar, `w_tld`, `q`, and `w` are column vectors, and `A` is a two-dimensional matrix. Thus, the subexpression $A^T * q$ is clearly the most expensive to compute, requiring $O(n^2)$ work to perform a matrix-vector product. However, the MATLAB interpreter will also compute a temporary matrix for the value A^T , requiring an additional n^2 space and copy operations, before it computes the product. Clearly, a temporary matrix should not be necessary to perform the computation. Unfortunately, the MATLAB language does not provide a way of expressing this computation as a single operation, thus forcing the evaluation of the subexpression. While a source-level transformation cannot directly avoid this, it can reduce the cost by realizing that $(q^T * A)^T$ is an equivalent and less expensive expression. Although this expression requires two transpose operations, in both cases vectors are transposed instead of matrices. Note the profile of the transformed code in Figure 6b.

The result is a better that twenty-fold increase on

that single statement and a three-fold increase on the entire benchmark. These kinds of transformations that exploit the semantics of matrix operations are not feasible at the C/FORTRAN level.

5 Comparison of Source-level Transformations and Compilation

In this section, we compare the separate and the combined effects of source-level and compiler optimizations. Figure 7 shows the performance benefits realized by different combinations of optimizations. The first three sets of bars represent the original MATLAB code, MCC compiled code with no unsafe optimizations, and MCC compiled code with all unsafe optimizations legal for that particular benchmark activated. The second three sets of bars represent the same measurements taken on source-level transformed code.

There are a number of interesting observations to be made. First, the performance improvement from source-level optimizations is quite comparable with performance improvement from MCC compiler optimizations. In four cases (CG, EC, RK, SOR), source-level optimizations have a roughly similar effect on performance when compared to safe MCC optimizations. In four other cases (FD, Ga, IC, QMR), source-level optimizations are better by a factor of two or more. In two of these cases (Ga, QMR), source-level optimizations outperform even unsafe optimizations by a wide margin! On the other hand, the remaining four cases (AQ, CN, Di, Mei) profit much more from compilation than source-level optimizations. These codes all contained expensive loops performing scalar operations that could not be eliminated by vectorization.

Second, source-level optimizations are best viewed as being complementary to compiler optimizations.

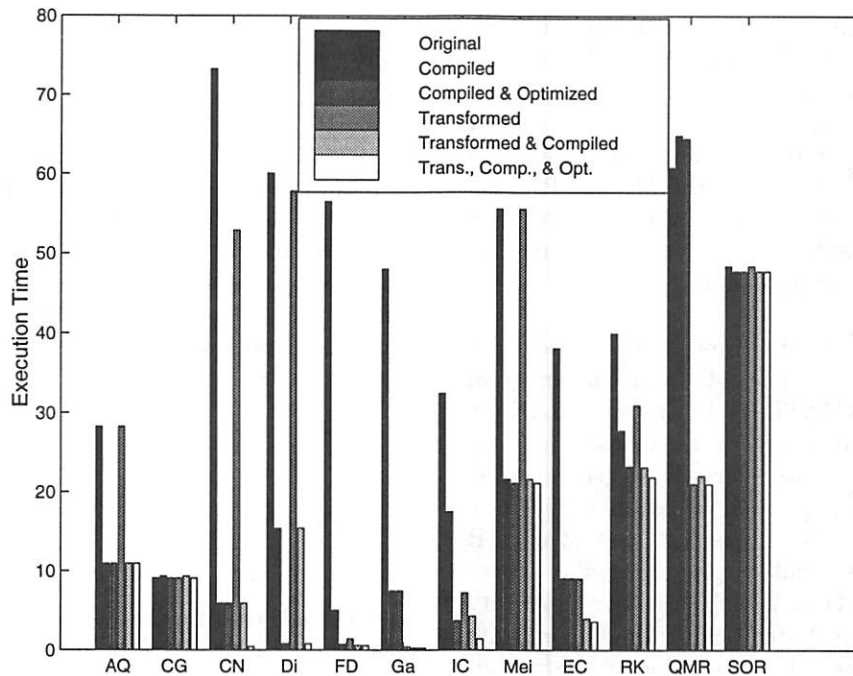


Figure 7: Source-Level Transformations and Compilation

The last two set of bars in Figure 7 illustrate the combined effect of source-level and compiler optimizations. For programs in which source-level transformations result in improved interpreted performance, these transformations result in improved compiled performance as well. Across all benchmarks, the combination of optimizations provides the best performance. In five cases (CN, FD, Ga, IC, EC), the combination significantly exceeds the effect of either source-level or compiler optimizations alone. Each of the different source-level transformations provides benefits not currently provided by MCC.

- When vectorization can generate high-level operations such as those in BLAS, as in Galerkin, efficient underlying libraries may be utilized by both interpreter and compiler. These libraries outperform code generated by a C or FORTRAN compiler on the corresponding loops.
- Although preallocation does not significantly effect the performance of every benchmark where it is applicable (see Figure 2b), it permits the otherwise unsafe compiler optimization of array bounds removal. The effect of this optimization after preallocation on, for example, Crank-Nicholson is dramatic.

- Algebraic optimizations such as the one used on QMR have no equivalent in MCC. These types of optimizations dependent on matrix properties cannot easily be applied at the lower level of a C compiler.

We conclude from these results that source-level transformations are key to obtaining good performance from MATLAB codes for both interpreted and compiled execution.

6 Related Work

As mentioned in the introduction, several MATLAB compilers have been developed or are under development. Each of these compilers translates MATLAB into a lower language such as FORTRAN, C, or C++.

There are two commercial MATLAB compilers. MCC [13] is from the The MathWorks, developers of MATLAB itself, and is the compiler studied in this paper. MCC can handle most features in MATLAB 5 and generates C code. MATCOM [9],

originally developed at the Israel Institute of Technology and now offered by MathTools, also handles most features of MATLAB 5 and generates C++ code. Both MCC and MATCOM are capable of generating either stand-alone programs or mex-files that may be linked back into the MATLAB interpreter. As far as we are aware, these are the only two publicly available compilers and the only two capable of generating mex-files.

There are a handful of compilers under development in academia. Falcon [4] from University of Illinois, translates MATLAB 4 into FORTRAN-90. Menhir [2], from Irisa in France, focuses on a re-targetable code generator capable of generating C or FORTRAN for sequential or parallel machines. MATCH [18], from Northwestern, uses MATLAB to directly target special purpose hardware. Three other compilers, CONLAB [5], from the University of Umea in Sweden, Otter [19], from Oregon State University, and one other from Northwestern University [20], explicitly target parallel machines by generating message passing code from MATLAB.

Of all of these compilers, only Falcon appears to consider source-level transformations [4, 11] along the lines of those described in this paper. However, these transformation must be applied interactively via a user tool and are not part of the automatic compilation process. Furthermore, the source-level transformations are limited to syntactic pattern match and replacement, so they do not provide a general solution for optimizations such as vectorization and preallocation. A similar type of idiom recognition appears to be performed by optimizing FORTRAN preprocessors such as KAPF [10] and VAST-2 [16]. These tools do attempt to detect matrix products in loop nests in order to generate BLAS operations. However, pattern matching is inherently limited in its ability to do this; neither processor is able to detect a vector-matrix-vector product written as in the Galerkin code shown in Section 4.1.

There has been compiler research on performing optimizations similar to the source-level transformations presented in this paper. Vectorization for vector supercomputers has been studied extensively over the past three decades, for example in [1, 21]. However, this work largely focuses on point-wise assignments and scalar operations between arrays and, occasionally, on reduction operations rather than the higher-level operations available in MATLAB. The problem of array bounds removal, similar to

preallocation and directly applicable to MATLAB, has been studied in the context of conventional languages [7].

Finally, a handful of projects [8, 15, 17] have developed parallel toolkits for use with the MATLAB interpreter. These toolkits allow MATLAB programs to directly access message passing libraries for inter-processor communication. To gain any performance benefit, users must parallelize their MATLAB programs using provided MATLAB-level message passing constructs.

7 Conclusion and Future Work

Source-level transformations provide an effective means of obtaining performance for MATLAB programs, regardless of whether they are interpreted or later compiled. These transformations are capable of eliminating many inefficiencies that currently available MATLAB compilers are unable to optimize away.

We have implemented an automatic tool to perform source-level optimizations as part of the FALCON project, a joint project of Cornell University and the University of Illinois, Urbana. A detailed description of this tool can be found in a companion paper [14]. Incorporating both source-level and lower-level optimizations into an interpreter in a just-in-time manner is an ongoing effort.

References

- [1] R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(2):491-542, October 1987.
- [2] S. Chauveau and F. Bodin. Menhir: An environment for high performance MATLAB. In *4th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Pittsburgh, PA, May 1998.
- [3] L. De Rose. *Compiler Techniques for MATLAB programs*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.

- [4] L. De Rose, K. Gallivan, E. Gallopoulos, B. Marsolf, and D. Padua. FALCON: A MATLAB interactive restructuring compiler. In *Languages and Compilers for Parallel Computing*, pages 269–288. Springer-Verlag, August 1995.
- [5] P. Drakenberg, P. Jacobson, and B. Kagstrom. A CONLAB compiler for a distributed memory multicomputer. In *6th SIAM Conference on Parallel Processing for Scientific Computing*, volume 2, pages 814–821, 1993.
- [6] J. Eaton. GNU Octave. <http://www.che.wisc.edu/octave>.
- [7] R. Gupta. A fresh look at optimizing array bound checking. In *Programming Languages, Design and Implementation*. ACM SIGPLAN, Jun 1990.
- [8] J. Hollingsworth, K. Liu, and P. Pauca. *Parallel Toolbox for MATLAB*. Wake Forest University, 1996. <http://www.mthsc.wfu.edu/pt/pt.html>.
- [9] Y. Keren. MATCOM: A MATLAB to C++ translator and support libraries. Technical report, Israel Institute of Technology, 1995.
- [10] Kuck and Associates, Inc. KAP for IBM Fortran and C. <http://www.kai.com/product/ibminf.html>.
- [11] B. Marsolf. *Techniques for the Interactive Development of Numerical Linear Algebra Libraries for Scientific Computation*. PhD thesis, University of Illinois at Urbana-Champaign, 1997.
- [12] The MathWorks, Inc. *How Do I Vectorize My Code?* <http://www.mathworks.com/support/tech-notes/v5/1100/1109.shtml>.
- [13] The MathWorks, Inc. *MATLAB Compiler*, 1995.
- [14] V. Menon and K. Pingali. High-level semantic optimization of numerical codes. In *International Conference of Supercomputing (ICS'99)*, June 1999.
- [15] V. Menon and A. Trefethen. MultiMATLAB: Integrating MATLAB with high performance parallel computing. In *Supercomputing*, November 1997.
- [16] Pacific Sierra Research Corporation. VAST-2 for XL Fortran. http://www.psrv.com/vast/vast_xlf.html.
- [17] S. Pawletta, T. Pawletta, and W. Drewelow. Comparison of parallel simulation techniques – MATLAB/PSI. *Simulation News Europe*, 13:38–39, 1995.
- [18] The MATCH Project. A MATLAB compilation environment for distributed heterogeneous adaptive computing systems. <http://www.ece.nwu.edu/cpdc/Match/Match.html>.
- [19] M. Quinn, A. Malishevsky, and N. Seelam. Otter: Bridging the gap between MATLAB and ScaLAPACK. In *7th IEEE International Symposium on High Performance Distributed Computing*, August 1998.
- [20] S. Ramaswamy, E. W. Hodges, and P. Banerjee. Compiling MATLAB programs to ScaLAPACK: Exploiting task and data parallelism. In *Proc. of the International Parallel Processing Symposium*, April 1996.
- [21] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1995.

Using Java Reflection to Automate Extension Language Parsing

Dale Parson (dparson@lucent.com)

Bell Laboratories, Lucent Technologies

Abstract

An extension language is an interpreted programming language designed to be embedded in a domain-specific framework. The addition of domain-specific primitive operations to an embedded extension language transforms that vanilla extension language into a domain-specific language. The LUXWORKS processor simulator and debugger from Lucent uses Tcl as its extension language. After an overview of extension language embedding and LUXWORKS experience, this paper looks at using Java reflection and related mechanisms to solve three limitations in extension language - domain framework interaction. The three limitations are gradual accumulation of ad hoc interface code connecting an extension language to a domain framework, over-coupling of a domain framework to a specific extension language, and inefficient command interpretation.

Java reflection consists of a set of programming interfaces through which a software module in a Java system can discover the structure of classes, methods and their associations in the system. Java reflection and a naming convention for primitive domain operations eliminate ad hoc interface code by supporting recursive inspection of a domain command interface and translation of extension language objects into domain objects. Java reflection, name-based dynamic class loading, and a language-neutral extension language abstraction eliminate language over-coupling by transforming the specific extension language into a run-time parameter. Java reflection and command objects eliminate inefficiency by bypassing the extension language interpreter for stereotyped commands. Overall, Java reflection helps to eliminate these limitations by supporting reorganization and elimination of hand-written code, and by streamlining interpretation.

1. Introduction

This paper examines the design of a set of Java utility interfaces and classes that simplify the work of integrating an extension language into a Java-based domain framework. Section 2 examines the role that extension languages play in extending domain-specific application frameworks. Section 3 looks at a commercial domain framework - extension language from Lucent called LUXWORKS. Experience in

designing, building and maintaining LUXWORKS has led to this current research project within Bell Labs. Section 3 examines some limitations that have surfaced in the original, C++-based implementation of LUXWORKS. Section 4 follows through by eliminating these limitations, using a combination of Java reflection, Java dynamic class loading, a language-neutral extension language abstraction, and a set of naming conventions. Section 5 looks at related extension language-Java efforts. Section 5 also summarizes.

2. Extension Languages for domain-specific software systems

An *extension language* is a programming language that extends a domain-specific software application, tool or framework (hereafter "framework"). Interpreted languages such as Scheme [1], Tcl [2], or Python [3] often serve as extension languages because their interpreters support interactive creation and execution of custom extensions by framework users at run time. Many proprietary, framework-specific extension languages have come and gone, but with the maturation of extension language technology and the mass acceptance of so-called *scripting languages* [4], there is now seldom a need to invent a new extension language for an interactive framework. The term *command language* highlights the fact that an extension language usually adds imperative commands to a framework.

An extension language provides an application programming interface (API) that supports connections between the extension language and the *system programming language* [4] such as C, C++ or Java that implements the basic capabilities of the domain-specific framework. An extension language supports three categories of extensions:

- A framework extends an extension language by adding *domain-specific primitives* to the extension language's instruction set.
- Conversely, an extension language extends a domain-specific framework by adding an interpreted language capability.
- An extension language user extends the composite framework-language by writing extension functions in the extension language.

A framework engineer codes domain-specific primitives in a system programming language for efficiency, for security, and for compatibility with existing code libraries. Extension language designers intend their languages to be extended with new primitives; extension languages differ in this way from languages whose definitions are frozen by standardization. A primitive becomes an integral part of the extension language. If the extension language supports *dynamic loading* of primitives, then even users in the field can extend a framework-language system. Otherwise framework developers must add primitives via static linking.

Figure 1 illustrates the major calling relationships in a framework-language system. A user interface or extension language program (a.k.a. "script") passes textual extension language expressions to the extension language's *eval* primitive, named after the classic LISP *eval* that evaluates a textual expression [5]. *Eval* tokenizes and parses the expression. *Eval* then executes the functional pieces of the expression by calling *apply* (again from the LISP legacy) with a function and its arguments as parameters. *Apply* calls a primitive function directly. *Apply* invokes a function written in the extension language by binding formal parameters to arguments and calling *eval* recursively with the text of the interpreted function. In extension languages that support incremental compilation of intermediate code (a.k.a. *byte code*), *apply* invokes a function written in the extension language by calling an intermediate code interpreter with the intermediate code of the compiled function.

In a purely functional, LISP-like language, the result returned from *eval* is the result of the outermost function invocation in the expression. Non-functional languages may include operators that do not reduce to primitive function calls; *eval* interprets these directly.

Eval builds atop generic extension language primitives as well as domain-specific primitives. Primitives build atop library classes and functions in their respective modules. In addition, domain code can call extension language primitives directly, without the overhead of *eval*-based parsing. Domain code can also call extension language library code, for example string or hash table utility functions. Finally, *eval* itself is a primitive, allowing nesting of expressions within expressions and within domain data structures to arbitrary depth. One typical use of nesting is attachment of an extension language *callback expression* to some condition in the domain framework. When the domain meets that condition, it triggers a *callback event* to the extension language, and the extension language evaluates the expression as part of domain execution. Event-directed control is found in graphical user interfaces, in simulation systems, in loosely coupled distributed computing, and in the JavaBeans programming environment [6].

3. Tcl and the LUXWORKS embedded system simulator and debugger

3.1 Tcl-Luxdbg architecture

The research project of this paper grew out of experience in the architecture, design, implementation and maintenance of Lucent's *luxdbg* simulator and debugger for embedded processors [7]. *Luxdbg* uses Tcl as its extension language [8]. For an overview of *luxdbg*'s architecture and design patterns see [9].

Figure 2 shows Tcl applied to *luxdbg*. *Luxdbg* registers the names and C++ function addresses of its primitives with Tcl at initialization time. Command strings drive Tcl. Tcl performs its language-specific manipulations on command strings, including variable substitution and concatenation of strings returned from nested Tcl

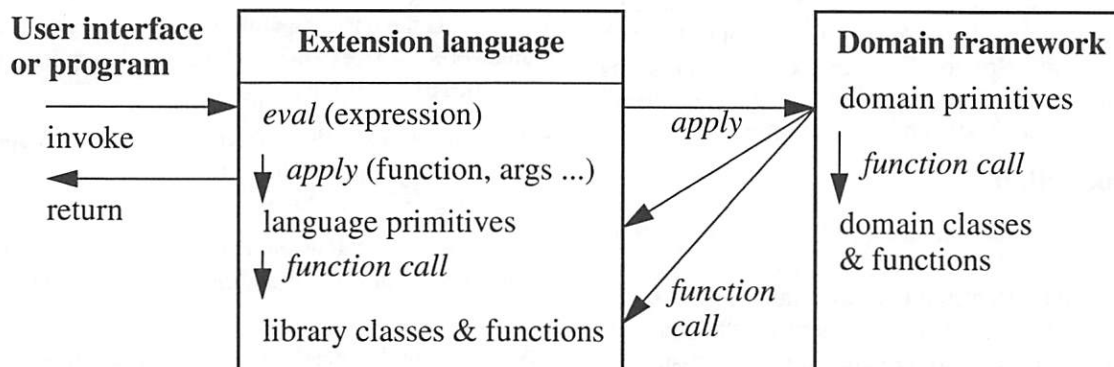


Figure 1: Calling patterns in a domain framework - extension language system

function invocations. Tcl then interprets byte codes for built-in primitives, and it forwards commands that start with registered primitive names to luxdbg. Like all Tcl functions, a luxdbg primitive returns a result string or error diagnostic upon completion of its invocation. Tcl can insert a result from a luxdbg primitive back into a higher-level expression; Tcl exception handling can catch luxdbg errors.

Figure 2 shows four classes of luxdbg primitives. *Processor management* primitives allow users to create, locate, initialize and destroy multiple processor instances. Processor instances include simulation models or connections to hardware processors of assorted types. *Processor access* primitives allow users to read and write processor state found in processor memory, registers and signals. *Processor control* primitives allow users to set and clear breakpoint event triggers, to handle breakpoint and error exceptions, to reset a processor, and to resume processor execution. *Processor IO* primitives allow users to connect input or output from processor input-output ports to data files or Tcl callback functions.

Exception processing and input-output processing provide two interesting examples of *event-driven callbacks* from the luxdbg domain framework to the Tcl extension language. A user can enable callbacks by associating a Tcl expression string with a breakpoint, with a processor error, or with a processor IO event. When one of these events occurs, luxdbg calls back to Tcl, passing a processor-event-expression triplet. Tcl evaluates the expression in the context of the processor and event. Callbacks can extend processor capabilities. An output callback, for example, can copy results from an output port to the input port of another processor, simulating interconnection. A breakpoint callback can

log debugging information. Callbacks can access multiple processors. All callbacks can conditionally continue or halt processor execution. Tcl callbacks can consequently implement a multiprocessor simulation scheduler.

These four categories of primitives combine with domain event-driven control to transform Tcl from a vanilla programming language to a domain-specific processor manipulation language. Tcl expressions evaluated within callback functions can extend or override the native computations of processors.

3.2 Tcl-Luxdbg limitations

Luxdbg as diagrammed in Figure 2 has a number of limitations.

3.2.1 Ad hoc primitive interface code

Connecting a new luxdbg primitive to Tcl requires writing new, special-case code. Tcl forwards commands to luxdbg through the following interface function:

```
int Primitive(ClientData clientData, Tcl_Interp *interp,
             int argc, char *argv[])
```

The *clientData* parameter is a C void pointer that the domain framework initializes when it registers a primitive with Tcl. Thereafter Tcl supplies this pointer as *clientData* when it calls the primitive. The pointer is useful for passing domain state information.

The *interp* parameter is a reference to the Tcl interpreter. It is useful for eval callbacks and calls to Tcl primitives and library functions.

The remaining two parameters, *argc* (argument count) and *argv* (argument vector), are standard fare for C programmers. Argument vector is a vector of strings and

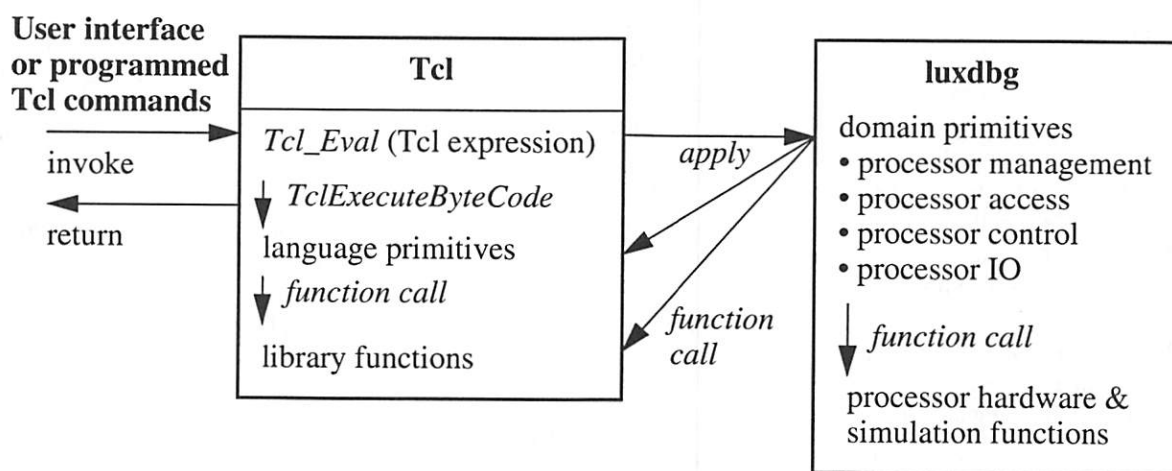


Figure 2: Calling patterns in the Luxdbg framework - Tcl language system

argument count holds its length. In calling a domain-specific primitive, Tcl stores the name and arguments of the primitive in `argv`. The primitive receives the arguments as strings, and the primitive has the responsibility of converting the strings to their appropriate values. Values might include strings as received, atomic types (e.g., int or float), Tcl-specific structured strings (e.g., Tcl lists), or domain-specific structured strings (e.g., an infix expression string for a processor debug statement). When a conversion error such as an invalid integer string occurs, the primitive has the responsibility of detecting the error and asserting a Tcl exception.

Tcl version 8.0 introduced an additional primitive interface for simplifying conversion of argument strings to atomic C types and for type mismatch detection [2]:

```
int TclPrim(ClientData clientData, Tcl_Interp *interp,
            int objc, Tcl_Obj *CONST objv[])
```

This interface replaces the string-based `argv` parameter with an array of *Tcl objects*. Client code can request conversion of one form of an object to another (e.g., a numeric string to an integer), but responsibility for directing argument type conversion and raising format exceptions still lies with the procedural code of primitives.

The need for testing parameter type-specific formats, directing translation of argument strings into objects of these types, verifying the correct number of arguments, and formatting return values, has resulted in a layer of luxdbg code to satisfy this need. Each new primitive requires a function to test and to convert its Tcl arguments to argument types required by its corresponding implementation function. The measurable cost is 1537 lines of non-comment code for 48 primitive functions in the current luxdbg, or about 32 lines of code on average for each primitive. While 1537 lines is a small number when compared with the roughly 80,000 lines of processor-neutral simulator and debugger code currently in luxdbg, the need to perform Tcl argument manipulation within each primitive contributes complexity out of proportion to the number of lines of code. Each designer of a primitive is saddled with the job of writing argument conversion code that has nothing to do with the primitive's semantics. Because this code comes in little, unrelated packets, each of which is specific to its primitive, it is written in an ad hoc manner. The process invites errors and time-consuming debugging and correction.

In addition, the creation of other tools that use Tcl as their extension language in our organization is causing the amount of code involved in ad hoc argc-argv

manipulation to grow. Section 4 shows that Java reflection and a simple naming convention for primitives can eliminate this code.

3.2.2 Hard-coded dependence on Tcl

Several research-oriented luxdbg users have expressed interest in using the Scheme and Python languages with luxdbg. Nothing about the architecture of Figure 2 precludes replacing Tcl with a different extension language, but there are some hurdles. ELK-Scheme [1] and Python [3] use the following primitive interfaces:

```
Object ELK_Constarg_Primitive(Object firstArg,
                               Object secondArg /*, etc. */)
```

```
Object ELK_Vararg_Primitive(int argc, Object *argv)
```

```
Object ELK_Lazyeval_Primitive(Object arglist)
```

```
int PyArg_ParseTuple(PyObject *argv, char *format,...)
```

ELK provides the first three, distinct interfaces. The first interface organizes arguments to a primitive call precisely by number of parameters supplied as part of primitive registration. The second interface allows a variable number of arguments. Both types use *eager evaluation*, resolving arguments to built-in Scheme types before calling the primitive. This is standard call-by-value evaluation. ELK passes arguments as dynamically-typed Scheme objects. The second interface supports variable-length argument lists and optional parameters. The third ELK interface uses *lazy evaluation* to pass a list of unevaluated Scheme objects to the primitive. This is a call-by-name mechanism that relies on the primitive to resolve argument text to argument values.

Python's `argv` is a *Python tuple* that contains call-by-value arguments. *Format* is a type conversion string similar to C's `scanf` function's format string [10]; it uses type conversion characters to direct type conversion of Python arguments into C atomic values and strings. *Format* implements a run-time interface definition language (IDL). Arguments following *format* are addresses of C variables that receive the results of format-directed conversions; there is no compiler or run-time check to ensure that format strings match the types of these variables. Python supports optional arguments and variable-length arguments, but `PyArg_ParseTuple`'s *format* is limited to a fixed maximum number of arguments.

This variety in extension language-to-primitive interfaces presents a problem for designing a multiple extension language interface for a domain framework. A unique domain interface might be required for each

extension language. There is an underlying similarity among these interfaces, however, that is of assistance. Each invokes a primitive function with an argument list of *extension language objects*. An object may be a *Tcl string*, a *Tcl_Obj*, an *ELK Object* or a *Python PyObject*, but it is some type of an extension language object. Multiple extension languages result in a two-dimensional type system, where the set of extension languages defines one dimension (set of types $type_x = \{Tcl, ELK, Python, \dots\}$) and the union of all extension language internal types defines the other (set of types $type_y = \{integer, float, string, sequence, \dots\}$). An object's type is an element of $type_x \times type_y$.

A simple solution would be to design yet another primitive functional interface, this one for the domain framework. Each extension language would require a language-primitive-to-domain converter ($type_x \times type_y \rightarrow type_{domain}$), that would map its proprietary object format into the domain object format, thereby eliminating the $type_x$ dimension. One problem with this approach is the need to design $type_{domain}$. Another problem is the overhead of constructing an intermediate copy of each object in this language-neutral format. Section 4 shows that Java incremental class loading and reflection can support automated ($type_x \times type_y \rightarrow type_{Jy}$) conversion, where $type_{Jy}$ represents Java classes \cup Java primitive types, and can include domain-specific classes. No intermediate format is necessary.

3.2.3 Unnecessary interpretation overhead

Every luxdbg command goes through extension language interpretation, resulting in execution that is slower than necessary.

In a scenario where a user enters commands directly into a terminal or commands come from an extension language script, interpretation overhead is necessary and acceptable. A user or program can supply any valid combination of commands, literal strings, control constructs (e.g., "if" statements), variable names and nested procedure calls. The extension language interpreter must resolve control flow, variable substitutions and return values from nested calls before calling domain primitive functions.

In luxdbg, however, typical interactive usage consists of a user interacting with a graphical user interface (GUI). The GUI forwards command strings to Tcl and receives result strings in reply. Tcl relays display update events from luxdbg to the GUI. Interpretation overhead is necessary in some cases, but for many cases it is not. Many of the commands coming from the GUI are *stereotyped commands*. These commands do not entail

extension language control constructs, variables or procedure calls. One example is a button for resuming processor execution; it always sends the "resume" command, which Tcl interprets and sends to a luxdbg primitive. Another example is a text box for modifying a processor register value; it always sends a register assignment command with a value entered by the user, and again Tcl interprets the command and sends it to luxdbg without changes. Most commands attached to GUI objects are stereotyped commands that Tcl passes to luxdbg unchanged. These could go directly from the GUI to luxdbg, avoiding Tcl overhead.

Unfortunately, the primitive function interface of luxdbg in Figure 2 is not uniform. Direct GUI-to-primitive function calls would entail detailed encoding, within the GUI, of parameter type signatures of all luxdbg primitive functions. Ad hoc code for connecting specific GUI buttons, menus and text boxes to specific primitive functions and parameters would proliferate. Tcl strings provide a uniform command medium that avoids this proliferation, albeit at the cost of interpretation overhead.

Once again the simple solution proposed for dealing with multiple extension languages suggests itself. A GUI could encode a stereotyped command as a function name and list of arguments using the common object format $type_{domain}$, and the domain framework would complete the job by mapping $type_{domain}$ types to primitive parameter types. Section 4 shows an extension language-neutral way to reuse any extension language's internal types as $type_{domain}$, thereby avoiding the creation of a framework-specific $type_{domain}$.

4. Java reflection supports an extension language-to-domain bridge

The limitations in the current implementation of luxdbg as well as opportunities afforded by Java infrastructure have set me on the path of replacing the upper, debugger and profiler layers of luxdbg with a new design in Java. Lower processor modeling and hardware interface layers remain in C++. Anticipated Java-enabled improvements include the following:

- Better networking support for distributed debugging.
- Dynamic configuration through incremental loading.
- Reflection-based extension language interface support.

This section looks at ways in which Java's incremental loading and reflection can help to overcome luxdbg's existing limitations.

4.1 Java reflection and a naming convention eliminate ad hoc primitive interface code

4.1.1 Java reflection

Reflection is a mechanism whereby parts of a software system can query the system itself, in addition to the usual ability to query for application domain information. Whereas domain queries are the typical queries of any query-supporting system, reflective queries are meta-queries [11]. Meta-queries ask the question: “How does this system do some particular thing?”

Reflection provides support for self-configuring tools and utilities. A generic tool or utility can read reflection information and adapt itself to its target system. JavaBeans provide a popular example [6]. A Java class that conforms to certain method naming conventions, and that may provide additional compiled information that describes the class, makes itself available for manipulation by graphical design tools. JavaBean system developers can instantiate objects, set object properties and create inter-object communication paths using tools that contain no encoding of the APIs or semantics of particular JavaBeans classes. Instead of hard coding target class dependencies, JavaBean tools encode knowledge of the JavaBean reflection API. At system design time these tools read JavaBean class-specific information through the API, giving customers access to the unique capabilities of each JavaBean class in a set of beans.

Java reflection allows Java code to query about available Java classes, interfaces, methods, fields, and their

properties at run time [12]. Figure 3 shows four major reflection classes — Class, Object, Method and Field — along with their associations and several methods that are important for this discussion. There are many more classes and methods in package `java.lang.reflect`.

Every Java object inherits from `java.lang.Object`. A reflection-based tool or utility calls `Object.getClass` to get a domain object’s `Class` (`java.lang.Class`). With the object’s `Class` in hand, a utility can determine constructors, superclass, implemented interfaces, nested classes, levels of protection, and most importantly for this discussion, methods and fields. `Class.getMethods` returns an array of `Method` objects, `Class.getField` returns a specific named `Field`, and `Class.getComponentType` returns the base type of an array. These are a few of the methods in class `Class`.

A `Method` supplies its name as a `String`, its parameter types as an array of `Class` objects, and its return type as a `Class` object. Clients of `java.lang.reflect` can build an array of correctly-typed `Object` arguments, and then call `Method.invoke` to invoke a `Method` on those arguments. `Method.getParameterTypes` provides the basis for automating the $(type_x \times type_y \rightarrow type_{Jy})$ conversion. `Method.invoke` provides the basis for automating domain primitive invocation.

A `Field` supplies its name as a `String`, its type as a `Class` object, and an assortment of `get` and `set` methods for retrieving and modifying its value. Reflective field query provides the basis for determining optional domain primitive parameters as part of automating the $(type_x \times type_y \rightarrow type_{Jy})$ conversion.

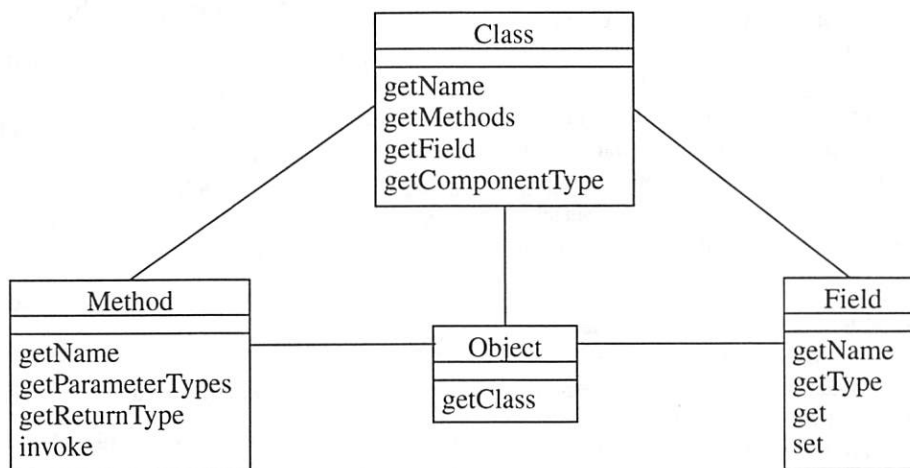


Figure 3: Central Java reflection classes

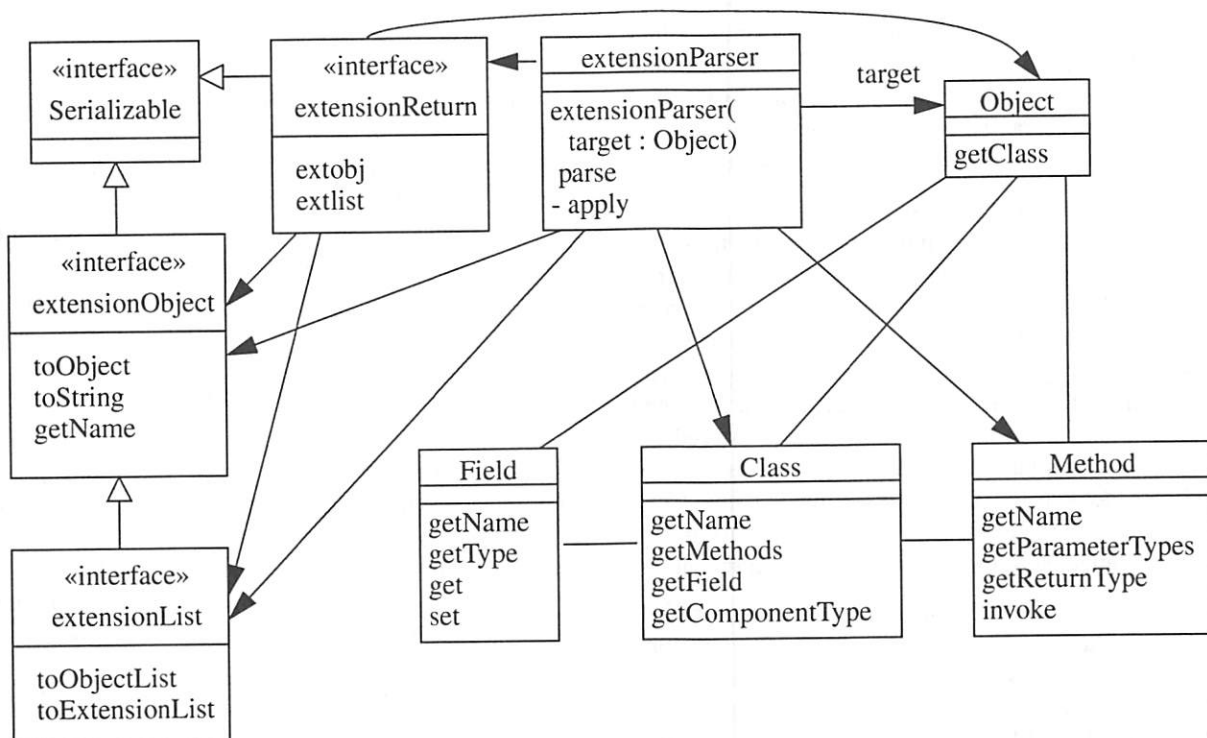


Figure 4: ExtensionParser translates extension language primitive calls

4.1.2 Eliminating ad hoc interface code

Figure 4 uses the Unified Modeling Language (UML) [13] to illustrate the major classes for $(type_x \times type_y \rightarrow type_{j_y})$ conversion and primitive method invocation. Hollow arrows signify inheritance, pointing from derived classes or interfaces to their parent classes or interfaces. Solid arrows are associations annotated for navigability. Clients navigate to the classes that serve them.

Class `extensionParser` is the central, client class of Figure 4.¹ `ExtensionParser` has three main methods: its *constructor*, its *parse* method, and its private *apply* method (“-” signifies private). The constructor takes a *target* `Object` as its parameter. The target is a domain object of any Java class type. Target makes domain primitives available to `extensionParser`. Each of target’s primitive methods adheres to the naming convention “command_NAME,” where NAME is the command name from an extension language’s perspective. For example in `luxdbg`,

String `command_stepi(int stepcount)`

is a target method that implements the “stepi” command for stepping a processor “stepcount” machine cycles, returning processor status as a `String` upon completion.

At construction time `extensionParser` uses `target.getClass` to get the domain object’s class, then it uses `Class.getMethods` to search the class for “command_” prefixed public methods. The constructor stores these in a hash table that is keyed on method name (without the “command_” prefix). Java permits method name overloading, and a slot in the hash table can hold multiple method references when that slot’s method name is overloaded.

Command parsing relies on the interfaces on the left side of Figure 4. A Java interface defines public method signatures, but it has no body. Another interface can extend an interface’s set of methods, and a class can implement the methods of any number of interfaces. In Figure 4 interface `extensionObject` represents an *extension language object* such as a `Tcl_Obj`, an `ELK Object` or a `Python PyObject`. Interface `extensionList` represents an ordered sequence of `extensionObjects`. Interface `extensionReturn` is a utility interface for converting Java objects into `extensionObjects` and `extensionLists`. `ExtensionReturn` is an inverse mapper

1. A working example of the code for `extensionParser` will be available as part of the on-line proceedings of this conference.

that formats return values when returning from a Java primitive to an extension language caller. The fact that these interfaces extend interface `java.io.Serializable` means that `extensionObjects`, `extensionLists` and `extensionReturns` can be passed as value parameters in networked method calls and stored as persistent objects.

Method `extensionParser.parse` receives an array of `extensionObjects` as an input parameter from an extension language primitive call. The first array element holds the command name, and the remaining elements hold its arguments. `ExtensionParser.parse` calls `extensionObject.getName` — all `extensionObjects` can be converted to `Strings` — and matches the returned command name to the method names stored in `extensionParser`'s hash table. A hash slot gives a list of candidate methods that match the command name. `Parse` also receives an `extensionReturn` object from the extension language for formatting `parse`'s return value.

The methods of `extensionObject` and `extensionList` come into play when `parse` calls `extensionParser.apply`. `Apply` is a recursive, backtracking match algorithm inspired by the more powerful match algorithms of PROLOG [14] and ML [15]. `Apply` takes as arguments the *command name*, a matching *candidate Method*, an input array of *extensionObject primitive-arguments*, another input array of *Method parameter types* (obtained in `parse` via `Method.getParameterTypes`), an array of *domain Object arguments* that `apply` populates by translating the `extensionObject` array, and `parse`'s *extensionReturn parameter*.

Each recursive call to `apply` attempts to match the next `extensionObject` argument to the next `Method` parameter type. `Apply` relies on an extension language-specific class that implements interface `extensionObject` to do the hard part. `extensionObject.toObject` takes the `Method` parameter type as its argument, and it attempts to convert itself into a Java `Object` of that type, returning that `Object` as its return value. Conversion starts by matching basic extension language object types to basic Java `Method` parameter types such as integers, booleans, floats and strings. When simple conversion fails, `apply` uses reflection to determine whether the `Method` parameter type has a class-static *valueOf* method that can convert an `extensionObject` into a domain `Object`. Target domain classes can provide custom `extensionObject-to-domain Object` converters by defining *valueOf*. `extensionObjects` transform into specialized domain-class objects without encoding domain awareness into `extensionObject` classes. Finally, if `toObject` cannot transform its `extensionObject` into the required domain `Object`, `toObject` throws a *typeMismatch* exception to `apply`.

A concrete `extensionObject.toObject` method is doing most of the work of $(type_x \times type_y \rightarrow type_{j_y})$ conversion. There is one such concrete method for each extension language in $type_x$, and by operating behind the abstract `extensionObject` interface it eliminates its specific $type_x$ language from `extensionParser`'s view. A concrete `extensionObject.toObject` method must be written once for each extension language. Its availability simplifies $(type_x \times type_y \rightarrow type_{j_y})$ conversion to $(type_y \rightarrow type_{j_y})$ conversion at the time that parameter matching occurs. The extension language internal type of $type_y$ maps itself to the domain type of $type_{j_y}$; the latter is `extensionObject.toObject`'s `Method` parameter type argument.

`ExtensionParser.apply` uses some mechanisms in addition to `extensionObject.toObject`. `Apply` checks whether the target `Method` accepts an `extensionObject` at the current position, passing an unaltered argument when acceptable. In this case the primitive must initiate conversion of the `extensionObject` at a later time. When a position match succeeds, `apply` advances to the next `extensionObject` and `Method` parameter type. When a position match fails, `apply` inspects a list of optional parameter positions built by `extensionParser`'s constructor. The constructor uses the target `Object`'s `Class.getField` method to locate an integer array "optional_NAME" for command method "NAME." If the array exists, each of its elements gives the offset of a parameter position that is optional for that command. `Apply` supplies a Java null reference for an optional position that cannot match the current `extensionObject`, then it continues searching.

If `apply` encounters an array `Method` parameter, it uses Java's *instanceof* predicate to determine whether its current `extensionObject` is in fact an `extensionList`, and `apply` uses reflection to determine whether the component type of the `extensionList` matches the component type of the parameter (obtained from `Class.getComponentType`). On a match, `apply` builds an array argument.

Finally, if `apply` arrives at the last `Method` parameter with a sequence of unmatched `extensionObjects`, and if the last `Method` parameter is an array, `apply` attempts to populate the array from these residual `extensionObjects`. A final array of type `extensionObject` receives the trailing arguments unaltered; `apply` invokes `extensionObject.toString` for a final array of type `String`.

If, at last, `extensionParser.apply` matches all parameters and consumes all `extensionObject` arguments, it uses `Method.invoke` to invoke the primitive method on its Java `Object` arguments. On success, `apply` uses `extensionReturn.extobj` to return the primitive result to

the extension language. When an applied method fails, apply and parse throw an exception back to the extension language.

Match failure, on the other hand, does not throw back to the extension language. Upon match failure, apply backtracks and attempts to use nulls for optional parameters; then it again works forward. If exhaustive attempts to match a particular Method fail, and if that Method name is overloaded, parse supplies another Method to apply. Only when all possibilities have been examined does parse report a usage error to the extension language.

Consider the example of Figure 5. This example shows two variants of a debugger *stop* command. Command “stop at location ?expression?” sets a breakpoint at a numeric processor address. Command “stop in function ?expression?” sets a breakpoint within a named function. Both primitives specify an optional callback expression to evaluate when the breakpoint fires. The callback is an interpreted extension language expression provided by a user.

This example is useful in pointing out what extensionParser both can and cannot do. It cannot distinguish a method on the basis of a textual keyword. Regardless of whether stop’s first argument is “at” or “in,” matching will pair it with the “String keyword” parameter of either method and attempt to use it. A more complex extensionParser mechanism would allow keyword-method pairs to be listed in a target Object field, and it would invoke a method only if its specified keywords are matched. It turns out that it is just as easy to have a matched method check its keywords on invocation, throwing exception *typeMismatch* on a keyword mismatch. ExtensionParser.apply treats *typeMismatch* as a mismatched method, and it continues searching for another match. This approach is similar to failure in the body of a PROLOG clause whose head has

matched its arguments [14].

In the first attempt for method “stop,” apply fails to match string “myfunc” to an int *location* parameter. Backtracking determines that the first parameter is not optional, and so apply backtracks to parse, which calls apply with another “stop” method. In this attempt, apply matches “myfunc” to the *function* parameter and the callback string to the *expr* parameter. Apply invokes the second *command_stop* method. If the callback expression had been missing, apply would have backtracked and inspected the optional parameter entry derived from *optional_stop_3*, filling *expr* with a null reference.

Note that while the callback expression {puts “stopped in myfunc” ; resume} is a Tcl list, nothing in the match algorithm encodes dependence on Tcl. The third extensionObject argument happens to be an extensionList that happens to be a Tcl list, but the concrete realization of this extensionObject as a Tcl list is unknown to extensionParser. The syntax handling for Tcl list construction occurs in the Tcl interpreter before the primitive call begins. Domain object access to list elements can occur through extensionList.toObjectList or extensionList.toExtensionList, converters that strip off language-specific list syntax and return an array of domain objects or an array of extension language objects respectively. In this example the domain object simply stores the Tcl list without decoding its values. When a domain processor reaches a breakpoint it passes the callback to the extension language (without encoding its identity as a Tcl interpreter) for evaluation.

This section has shown how Java reflection and a naming convention on method names eliminates hand coded parameter conversion code for primitive methods. ExtensionParser aligns parameters and reports errors. The next section shows how dynamic class loading combines with the reflective capabilities of

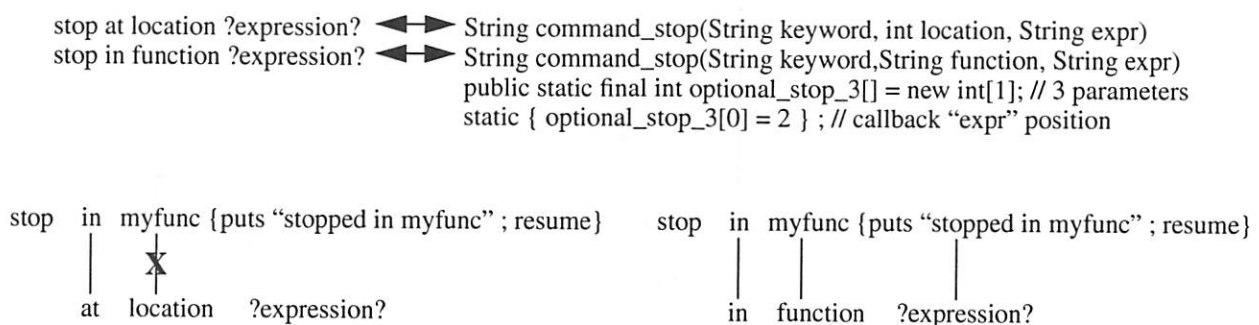


Figure 5: An example of automatic primitive method parameter alignment

extensionObject to support extension language selection at run time.

4.2 Extension language as a parameter

Figure 6 shows interfaces *extensionObject*, *extensionList* and *extensionReturn* of Figure 4, and it also shows interface *extensionLang* that encapsulates an extension language interpreter. *ExtensionLang* includes public methods for interpreting expressions and applying functions. Figure 6 also shows *Tcl* implementation classes that implement these four interfaces. The dashed lines signify UML *realization* equivalent to Java's *implements* directive. *TclObject*, *TclList* and *TclReturn* assist in converting *Tcl* objects to Java Objects and in returning Java Objects to *Tcl* as discussed in the last section. *TclInterp* houses a *Tcl* interpreter that uses *TclObjects*, *TclLists* and *TclReturn* to communicate with Java primitive methods.

The *Tcl* classes of Figure 6 support *Tcl* by wrapping the C implementation of *Tcl* 8.1.1 with Java Native Interface (JNI) proxy methods [16]. Each proxy method calls its *Tcl* counterpart through JNI's C binding. *Tcl* does not encode dependence on Java, and most Java

classes that use extension languages encode dependence only on the abstract interfaces of Figure 6.

The four classes *TclInterp*, *TclObject*, *TclList* and *TclReturn* constitute a *Tcl software component*. Collectively they implement the four interfaces needed to install an extension language in this Java framework. Furthermore, Java's *ClassLoader.loadClass* method allows this Java framework to load a specific extension language, by name, at run time. The *luxdbg* extension language loader appends the string "Interp" to the extension language name (e.g., "TclInterp"), loads that specific language class from the *luxdbg* package that houses extension language components, and performs a run-time type check to ensure that the loaded class implements the *extensionLang* interface. *ExtensionLang* uses the other interfaces of Figure 6, and loading an extension language also loads the other concrete classes. Loading *TclInterp* loads *TclObject*, *TclList* and *TclReturn* classes as well. The loader is similar to reflection in supporting a string-based approach to determining available extension languages at run time. A GUI could inspect available languages in *luxdbg*'s extension language package and allow a user to select

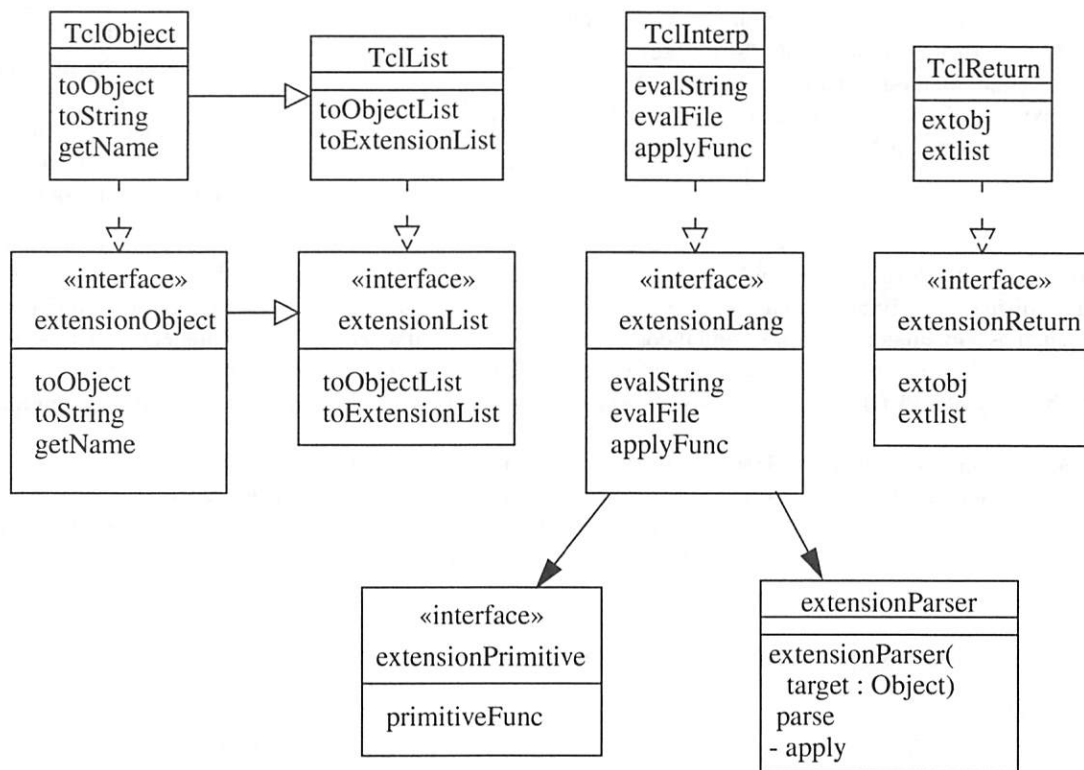


Figure 6: ExtensionLang encapsulates an extension language

the language of choice.

ExtensionLang defines helper interface *extensionPrimitive* that specifies method *primitiveFunc*. *ExtensionPrimitive.primitiveFunc* has the same signature as *extensionParser.parse*. For the *TclInterp* example, *TclInterp* queries its *extensionParser* for “command_” primitive names at construction time, and it registers each primitive command with its C-level *Tcl* interpreter. Registration includes a pointer to a JNI function that, when called as a primitive from *Tcl*, calls *TclInterp*’s version of *extensionPrimitive.primitiveFunc*. *Tcl* delegates primitive commands to a C-level JNI function, which in turn delegates to *TclInterp*’s *extensionPrimitive.primitiveFunc*, which in turn delegates to *extensionParser*, which then performs the matching and method invocation discussed in the last section. Return values and exceptions come back the delegation chain. *TclReturn* converts return objects to *Tcl* objects, and the JNI function converts Java exceptions to *Tcl* exceptions.

Thus, four straightforward interface abstractions — an extension language, its objects, object sequences, and return values — suffice to encapsulate a complex language as a Java software component. The *type_x* term has become a run-time parameter that users can set.

4.3 Commands that bypass the interpreter

Section 3.2.3 raised the issue of unnecessary interpretation overhead. Stereotyped commands from a GUI need not go through an extension language interpreter because the extension language does not change stereotyped command strings. The alternative of connecting a GUI directly to domain object implementation methods is undesirable because it over-couples GUI code to primitive method signatures. GUI code becomes ad hoc. Section 3.2.3 proposed that a GUI could encode a stereotyped command as a function name and list of arguments using the common object format *type_{domain}*, and the domain framework would complete the job by mapping *type_{domain}* types to primitive parameter types.

ExtensionParser.parse is precisely the method needed to provide a uniform command interface that bypasses the extension language interpreter. *Parse*’s main input parameter is an array of *extensionObjects* to translate. A Java GUI can map user interface events (e.g., button pushes, etc.) → arrays of *Strings*, then map *Strings* → *type_{domain}* objects in the form of *extensionObjects* by calling *extensionReturn.extobj*, then send an array of *extensionObjects* to *extensionParser.parse*; *parse* then maps *type_{domain}* objects as *extensionObjects* → *type_{Jy}* via *extensionParser.apply* and *extensionObject.toObject*.

All *extensionObject* classes can hold strings (string representation is possible for all extension language types), so mapping *Strings* → *type_{domain}* objects entails no type conversion overhead.

Going back to Figure 1, a User Interface can bypass the Extension Language component and send all stereotyped commands directly to the Domain Framework’s *extensionParser* interface. Two design patterns from the Gang of Four book are conspicuous here [17]. *ExtensionParser* implements the *Facade Pattern*. *ExtensionParser* provides a unified, homogeneous interface to a set of heterogeneously typed primitive domain methods. Next, *extensionParser.parse*’s input array of *extensionObjects* implements the *Command Pattern*. The first array element is a command name and the remaining elements are its arguments. Command arrays can be stored, queued, forwarded and ultimately executed via *extensionParser.parse*.

The only potential drawback to bypassing the extension language is the fact that users cannot extend or otherwise redefine stereotyped commands in the extension language if those commands always bypass the extension language. *Luxdbg* avoids this problem by registering primitives, including stereotyped command names, with the extension language, and then having the extension language notify UI components if any stereotyped commands are redefined. At that point those commands are no longer stereotyped. After redefinition, UI components must send these commands through the extension language component.

4.4 Performance

The current Java implementation of *luxdbg* transforms the C++ implementation of Figures 1 and 2 into associations of UI and Domain Framework Java components interconnected by *extensionLang*, *extensionObject*, *extensionList*, *extensionReturn* and their *Tcl* concrete counterparts. UI-Domain communications ultimately pass through *extensionParser* to *luxdbg*’s Domain Framework. How much does all this encoding, message passing, and reflection-based decoding cost?

The answer is that, compared to extension language interpretation costs, reflection-based command parsing is cheap. Table 1 summarizes the results of sending 250,000 command calls through each of three interfaces:

- direct UI-to-Domain object method calls with no extension language or *extensionParser* involvement
- construction and passage of command objects (i.e.,

extensionObject arrays) from UI strings to extensionParser as discussed in Section 4.3, bypassing the extension language interpreter

- evaluation of UI command strings in the extension language interpreter, which uses extensionParser to decode primitive calls

The target primitive method takes a single integer parameter and it returns a constant Java String. The machine is a lightly loaded Toshiba Tecra laptop with a 266 MHz Pentium processor, 96 Mbyte RAM and 32Kbyte internal cache, running Windows 95, Sun's Java Development Kit 1.2 and Tcl 8.1.1. The test driver invokes Java's garbage collector immediately before each of the three measured 250,000-call series. Table 1 reports time-per-call in microseconds.

Table 1: μ Seconds-per-call for direct calls, command objects and interpreted expressions

test	direct	parsed command objects	Tcl 8.1.1 interpreter
argv, 1 method	0	42	399
argv, 50 methods	0	29	378
Tcl_Obj, 1 method	0	43	417
Tcl_Obj, 50 methods	0	36	391

The first two rows use Tcl's original string-based, char **argv primitive interface. The last two rows use the newer Tcl_Obj object interface that attempts to keep objects in an appropriate primitive format (e.g., string, int or float) until the object is needed. The first and third rows define only 1 method, the test target method, in the test domain object. The second and fourth rows define 50 primitive methods, including 2 additional, overloaded instances of the target method name with different parameter types.

All rows show that 250,000 calls were not enough to bring direct call overhead out of the noise. The 0 figure does signify that communication and interpretation overhead accounts for all measurable delays in other

columns.

Average extension language interpretation runs about 10.6 times slower than command objects that bypass the interpreter. Clearly overhead is eliminated. At 29 to 43 microseconds of command overhead per GUI event, command objects that bypass the extension language are clearly fast enough. There is no reason to over-couple the GUI to the Domain Framework for speed.

378 to 417 microseconds of interpreter overhead includes the call to extensionParser.parse on the Domain Framework side of the extension language. Roughly 400 microseconds per call is still not a lot of overhead. C++ luxdbg has the additional problem that the extension language extracts all Domain Framework-to-GUI update events and sends them to the GUI. Java luxdbg will eliminate the extension language from stereotyped GUI callbacks as well.

The first surprise comes with the fact that the second row, working with a more heavily populated extensionParser Method hash table than the first row, is nevertheless faster than the first row. This result was consistent across tests, and it is repeated between rows three and four. The only conclusion is that Java hash tables are marginally more efficient when populated with a typical command set size.

The next surprise comes with the fact that the first two rows, last column, are marginally more efficient than their counterparts in the last two rows. The char **argv implementation of Tcl objects in the first two rows translates Tcl objects to Java strings immediately upon leaving Tcl to invoke a primitive. The Tcl_Obj implementation of Tcl objects in the last two rows stores a reference to a C-level Tcl_Obj in each Java TclObject. It does not translate a Tcl_Obj into a domain Object until TclObject.toObject runs, and it converts it directly to the integer needed by the test method. It skips the intermediate format of a Java String. The benefit of avoiding the intermediate String format appears to be offset by the fact that TclObject.toObject must call through the Java Native Interface to C in order to extract the integer value by calling Tcl's Tcl_GetIntFromObj library function. JNI calls add overhead. The TclObject caches its value to avoid subsequent calls through JNI, but typical extensionObjects (e.g., TclObjects) require only one toObject call, so caching is not much help. Tcl's original char **argv is both simpler to program and faster for this application.

ExtensionParser cost is roughly one tenth the cost of extension language + extensionParser costs for calling a stereotyped command that does nothing. ExtensionParser's percentage contribution to overhead

diminishes as the extension interpreter is given real scripts to interpret, and as the Domain Framework is given real primitives to execute. Interpreter and domain costs go up while extensionParser costs remain constant. Clearly performance is adequate for extension language primitive interfaces and Command design pattern objects.

5. Related work and conclusions

5.1 Related work

Other existing Java-based implementations of Tcl include Jacl and TclBlend [18]. Jacl is a partial implementation of the Tcl interpreter in Java, while TclBlend is a conventional C Tcl implementation with an interface to Java. The report on Jacl and TclBlend states, "For the Java platform, we envision an architecture that includes Java as the 'component' language used by component developers and Tcl as the 'glue' language used by application assemblers." [18] That perspective appears to make Tcl the center of the framework. Tcl is responsible for interconnecting and synchronizing Java components. Jacl is purported to be slow [19] — Jacl interprets Tcl on top of a Java bytecode interpreter — although a Jacl implementation that produces Java bytecodes is certainly possible.

Luxdbg's use of Java and Tcl takes a different approach. Luxdbg uses Tcl where extension language interpretation makes sense, but it does not put extension language interpretation overhead in the middle of the architecture. Luxdbg's extension language interpreter has access to all primitives, and it is possible to coordinate all framework activities from Tcl, but it is not mandatory. JavaBeans construction environments do a reasonable job of generating "glue" code for stereotyped component interactions, saving the extension language for what it does best, extending the system at run time. Given Tcl's inability to coordinate multiple Java threads within a single Tcl interpreter [20], Tcl is not an ideal candidate for the center of any multi-threaded Java framework.

Part of the impetus for creating the extension language-neutral extensionLang interface was the desire to experiment with JPython [21], a Java implementation of Python, within luxdbg. Interesting features of JPython include dynamic compilation to Java bytecodes for performance, and the ability to extend existing Java classes in JPython. JPython is tightly integrated into Java, and implementing extensionLang and the related interfaces should be straightforward. This is an area for future investigation.

5.2 Conclusions

This paper started out as an overview of using extension language components with application domain frameworks. It looked at a particular framework, luxdbg, and its coupling to the Tcl extension language. Integration of Tcl into C++ luxdbg has been a great success, but it has suffered from a few limitations. Interface code from Tcl to C++ primitives is often ad hoc and annoying to program, tight coupling of the luxdbg framework to Tcl limits its ability to work with other extension languages, and putting the extension language in the center of all UI-to-Domain interactions adds unnecessary overhead.

Java reflection and dynamic loading have provided the basis for a set of mechanisms that overcome these limitations. Reflection and a naming convention allow class extensionParser and interface extensionObject to work together to eliminate ad hoc primitive interface code. The abstract extensionLang interface and dynamic, name-based class loading work together to make the specific extension language in a system a run-time parameter. Command pattern objects in the form of extensionObject arrays support stereotyped GUI-to-Domain interactions that are uniform and efficient. Clearly Java reflection and dynamic loading are very powerful tools for enhancing the utility of extension languages.

6. References

1. "The Extension Language Kit (ELK)", <http://www-rn.informatik.uni-bremen.de/software/elk/>. ELK is an implementation of Scheme organized for use as an extension language.
2. Brent Welch, *Practical Programming in Tcl and Tk*, Second Edition. Upper Saddle River, NJ: Prentice Hall PTR, 1997.
3. Guido van Rossum, *Extending and embedding the Python interpreter*. Amsterdam: Stichting Mathematisch Centrum, 1995, also at <http://www.python.org/doc/ext/ext.html>.
4. John Ousterhout, "Scripting: Higher Level Programming for the 21st Century," *IEEE Computer*, March, 1998, or Scriptics Corporation, <http://www.scriptics.com/people/john.ousterhout/scripting.html>.
5. John Allen, *Anatomy of LISP*. New York: McGraw-Hill, 1978.
6. Robert Englander, *Developing Java Beans*. Sebastopol, CA: O'Reilly, 1997.
7. *LUXWORKS Debugger User Guide*, luxdbg Version 1.7.0, Lucent Technologies, December, 1998.

8. D. Parson, P. Beatty and B. Schlieder, "A Tcl-based Self-configuring Embedded System Debugger." Berkeley, CA: USENIX, *The Fifth Annual Tcl/Tk Workshop '97 Proceedings*, Boston, MA, July 14-17, 1997, p. 131-138.
9. D. Parson, P. Beatty, J. Glossner and B. Schlieder, "A Framework for Simulating Heterogeneous Virtual Processors." Los Alamitos, CA: IEEE Computer Society, *Proceedings of the 32nd Annual Simulation Symposium*, IEEE Computer Society / Society for Computer Simulation International, San Diego, CA, April, 1999, p. 58-67.
10. Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Second Edition. Englewood Cliffs, NJ: Prentice Hall, 1988.
11. Frank Buschmann, "Reflection," in *Pattern Languages of Program Design 2*, ed. J. Vlissides, J. Coplien and N. Kerth, Reading, MA: Addison-Wesley, 1996, p. 271-294.
12. Ken Arnold and James Gosling, *The Java™ Programming Language*, Second Edition. Reading, MA: Addison-Wesley, 1998.
13. James Rumbaugh, Ivar Jacobson and Grady Booch, *The Unified Modeling Language Reference Manual*, Reading, MA: Addison-Wesley, 1999.
14. W. F. Clocksin and C. S. Mellish, *Programming in PROLOG*, Second Edition. Berlin: Springer-Verlag, 1984.
15. Jeffrey D. Ullman, *Elements of ML Programming*. Englewood Cliffs, NJ: Prentice-Hall, 1994.
16. Rob Gordon, *Essential JNI: Java Native Interface*. Upper Saddle River, NJ: Prentice Hall, 1998.
17. E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.
18. Ray Johnson, "Tcl and Java Integration," Sun Microsystems Laboratories, February 3, 1998. See <http://www.scriptics.com/java/>
19. Assorted discussions on comp.lang.tcl.
20. Tcl 8.1.1 documentation at <http://www.scriptics.com>
21. JPython home page at www.jpython.org

DSL Implementation Using Staging and Monads

Tim Sheard, Zine-el-abidine Benaissa, and Emir Pasalic
Pacific Software Research Center
Oregon Graduate Institute
P.O. Box 91000 Portland, Oregon 97291-1000 USA
sheard@cse.ogi.edu, <http://cse.ogi.edu/~sheard>

Abstract

The impact of Domain Specific Languages (DSLs) on software design is considerable. They allow programs to be more concise than equivalent programs written in a high-level programming languages. They relieve programmers from making decisions about data-structure and algorithm design, and thus allows solutions to be constructed quickly. Because DSL's are at a higher level of abstraction they are easier to maintain and reason about than equivalent programs written in a high-level language, and perhaps most importantly *they can be written by domain experts rather than programmers*.

The problem is that DSL implementation is costly and prone to errors, and that high level approaches to DSL implementation often produce inefficient systems. By using two new programming language mechanisms, program staging and monadic abstraction, we can lower the cost of DSL implementations by allowing reuse at many levels. These mechanisms provide the expressive power that allows the construction of many compiler components as reusable libraries, provide a direct link between the semantics and the low-level implementation, and provide the structure necessary to reason about the implementation.

1 Introduction

We outline an improved *method* for the design and implementation of Domain-Specific Languages (DSLs). The method builds upon our experience with staged programming using the staged programming language METAML [27, 26]. The method also

incorporates ideas from other researchers in the areas of modular language design [28, 24, 12], correct compiler generation [15, 19, 18, 16, 10], and partial evaluation [8, 13]. While relying on recent advances in functional programming (such as higher-order type constructors, and local polymorphism), it is applicable to *all kinds of languages*, not just applicative ones. The method unifies many of these ideas into a coherent process.

A problem with the DSL approach to software construction is its cost. Realizing a DSL requires an implementation. Such implementations are large and expensive to produce. So, unless many solutions are required, it may not pay to build a compiler or other implementation mechanism. DSL implementation is also conceptually hard. Most software engineers are not comfortable taking on the task of language design and implementation. Even if they are, language implementation is a difficult, complex process that does not easily scale. An implementation for a simple language often does not scale as the language evolves to meet newer demands. Lowering the cost of DSL implementations, and making good ones more manageable, will make the DSL approach applicable to a broader domain of problems.

Our approach to solving these problems is to apply new methods of abstraction such as monads [28, 31] and staging [27, 26] to the implementation of DSLs. This makes the effort required to build a compiler for a DSL reusable and spreads the cost over several DSLs. To make language implementation manageable for the masses, there must exist good rules of thumb for language implementation. One way to accomplish this is by elaborating a *step by step method* that splits the labor into well-defined steps, each with a relatively small amount of work. In our method, each step deals with an orthogonal design decision. By using good abstraction principles, our method partitions each design decision into a sepa-

rate code module. In addition, our method makes explicit the propositions that must be proved to show the correctness of the compiler with respect to its semantics.

Our method comprises the following steps. First, construct the denotational semantics as an interpreter in a functional language. Second, capture the effects of the language, and the environment in which the target language must run, in a monad. Then rewrite the interpreter in a monadic style. Third, stage the interpreter using meta-programming techniques. This staging is similar to the staging of interpreters using a partial evaluator, but is explicit rather than implicit, since the programmer places the annotations directly, rather than using an automatic binding time analysis to discover where they should be placed. This leaves programmers in complete control, and they can limit what appears in the residual program. Fourth, the resulting program is both a data-structure and a program, so it can be both directly executed and analyzed. This analysis can include both source to source transformations, or translation into another form (i.e. intermediate code or assembly language). Because the programmer has complete control over the earlier steps, the structure of the residual program is highly constrained, and this final translation can be a trivial task.

Staging of interpreters using partial evaluation has been done before [1, 5]. The contribution of this paper is to show that this can all be done in a single program. A system incorporating staging as a first class feature of a language is a powerful tool. While using such a tool to write a compiler the source language can be given semantics, it can be staged, translated, and optimized all in a single paradigm. It requires neither additional processes nor tools, and is under the complete control of the programmer; all the while maintaining a direct link between the semantics of interpreter and those of the compiler.

2 Staging in MetaML

METAML is almost a conservative extension of Standard ML. Its extensions include four staging annotations. To delay an expression until the next stage one places it between meta-brackets. Thus the expression `<23>` (pronounced "bracket 23") has type

`<int>` (pronounced "code of int"). The annotation, `~e` splices the deferred expression obtained by evaluating `e` into the body of a surrounding Bracketed expression; and `run e` evaluates `e` to obtain a deferred expression, and then evaluates this deferred expression. It is important to note that `~e` is only legal within lexically enclosing Brackets. We illustrate the important features of the staging annotations in the short METAML sessions below.

```
-| val z = 3+4;
val z = 7 : int
```

Users access METAML through a *read-type-eval-print* top-level. The declaration for `z` is read, type-checked to see that it has a consistent type (`int` here), evaluated (to 7), and then both its value and type are printed.

```
-| val quad =
  ( 3+4, <3+4>, lift (3+4), <z> );
val quad =
  ( 7, <3 % 4>, <7>, <%z> ) :
  ( int * <int> * <int> * <int> )
```

The declaration for `quad` contrasts normal evaluation with the three ways objects of type code can be constructed. Placing brackets around an expression (`<3+4>`) defers the computation of `3+4` to the next stage, returning a piece of code. Lifting an expression (`lift (3+4)`) evaluates that expression (to 7 here) and then lifts the value to a piece of code that when evaluated returns the same value. Brackets around a free variable (`<z>`) creates a new constant piece of code with the value of the variable. Such constants print with a `%` sign to indicate they are constants. We call this *lexical-capture* of free variables. Because in METAML operators (such as `+` and `*`) are also identifiers, free occurrences of operators in constructed code often appear with `%` in front of them.

```
-| fun inc x = <1 + ~x>;
val inc = Fn : ['a].<int> -> <int>
```

The declaration of the function `inc` illustrates that larger pieces of code can be constructed from smaller ones by using the escape annotation. Bracketed expressions can be viewed as *frozen*, i.e. evaluation does not apply under brackets. However, it is often convenient to allow some reduction steps inside a

large frozen expression while it is being constructed, by “splicing” in a previously constructed piece of code. METAML allows one to *escape* from a frozen expression by prefixing a sub-expression within it with the tilde (~) character. Escape must only appear inside brackets.

```
-| val six = inc <5>;
val six = <1 %+ 5> : <int>
```

In the declaration for `six`, the function `increment` is applied to the piece of code `<5>` constructing the new piece of code `<1 %+ 5>`.

```
-| run six;
val it = 6 : int
```

Running a piece of code, strips away the enclosing brackets, and evaluates the expression inside. To give a brief feel for how MetaML is used to construct larger pieces of code at run-time consider:

```
-| fun mult x n =
    if n=0 then <1>
    else <~x * ~(mult x (n-1))>;
val mult = fn : <int> -> int -> <int>

-| val cube = <fn y => ~(mult <y> 3)>;
val cube = <fn a => a * (a * (a * 1))>
          : <int -> int>

-| fun exponent n = <fn y => ~(mult <y> n)>;
val exponent = fn : int -> <int -> int>
```

The function `mult`, given an integer piece of code `x` and an integer `n`, produces a piece of code that is an `n`-way product of `x`. This can be used to construct the code of a function that performs the `cube` operation, or generalized to a generator for producing an exponentiation function from a given exponent `n`. Note how the looping overhead has been removed from the generated code. This is the purpose of program staging and it can be highly effective as discussed elsewhere [4, 6, 11, 23, 27]. In this paper we use staging to construct compilers from interpreters.

3 Monads in Language Design

We make significant use of the notion of monads. A good way to think of a monad is as an abstract

datatype that captures the side effects and actions inherent in the language being translated in the methods of the abstract datatype. An important feature of a monad is that also describes, in a purely functional way, how these effects and actions interact. Like any good abstract datatype, we are free to implement the actions in any way we want as long as our implementation behaves like its purely functional description.

The ultimate efficiency of the compiler depends on making good use of the low-level primitives of the target language. Monads are the glue that we use to tie high-level (purely functional) descriptions of languages to the low-level implementation features of the target environment.

A monad is a type constructor M (a type constructor is a function on types, which given a type produces a new type), and two polymorphic functions $unit : a \rightarrow M(a)$ and $bind : M(a) \rightarrow (a \rightarrow M(b)) \rightarrow M(b)$. The way to interpret an expression with type $M(a)$ is as a computation that represents a potential *action* and that also returns a value of type a .

An action might perform I/O, update a mutable variable, or raise an exception. One can implement a monad in a purely functional setting by emulating the actions. This is done by explicitly threading “stores”, “I/O streams”, or “exception continuations” in and out of all computations. We call such an emulation the *reference implementation*. Using a functional implementation allows equational reasoning about the reference implementation, however it is usually quite inefficient.

The two polymorphic functions $unit$ and $bind$ must meet the following three axioms:

$$\begin{aligned} \text{(left id)} \quad & bind (unit\ x) (\lambda y.e) = e[x/y] \\ \text{(right id)} \quad & bind\ e (\lambda y.unit\ y) = e \\ \text{(bind assoc)} \quad & bind (bind\ e (\lambda x.f)) (\lambda y.g) = \\ & bind\ e (\lambda z.bind\ (f[z/x])(\lambda w.g[w/y])) \end{aligned}$$

where $e[x/y]$ is the result of the substitution of the free occurrences of the variable x in e by the variable y .

The monadic operators, $unit$ and $bind$, are called the standard morphisms of the monad. The $unit$ operator takes a pure value and turns it into an empty action. The $bind$ operator sequences two actions. A useful monad will also have non-standard

morphisms that describe the primitive actions of the monad (like *read* the value from a variable and *write* a variable in the monad of mutable state).

For more background on the use of monads see [29, 31, 30].

4 Monads in METAML

In METAML a monad is a data structure encapsulating a type constructor *M* and the *unit* and *bind* functions.

```
datatype ('M : * -> * ) Monad = Mon of
  (* unit function *)
  ([ 'a ]. 'a -> 'a 'M) *
  (* bind function *)
  ([ 'a, 'b ]. 'a 'M -> ('a -> 'b 'M) -> 'b M);
```

This definition uses SML's postfix notation for type application, and two non-standard extensions to ML. First, it declares that the argument (*'M : * -> **) of the type constructor *Monad* is itself a unary type constructor [7]. We say that *'M* has *kind*: ** -> **. Second, it declares that the arguments to the constructor *Mon* must be polymorphic functions [17]. The type variables in brackets, e.g. [*'a, 'b*], are universally quantified. Because of the explicit type annotations in the *datatype* definitions the effect of these extensions on the Hindley-Milner type inference system is well known and poses no problems for the METAML type inference engine.

In METAML, *Monad* is a first-class, although *pre-defined* or *built-in* type. In particular, there are two syntactic forms which are aware of the *Monad* datatype: *Do* and *Return*. *Do* and *Return* are METAML's syntactic interface to the *unit* and *bind* of a monad. We have modeled them after the denotation of Haskell[9, 20]. An important difference is that METAML's *Do* and *Return* are both parameterized by an expression of type *'M Monad*. *Do* and *Return* are syntactic sugar for the following:

<pre>(* Syntactic Sugar</pre>	<pre>Derived Form *)</pre>	
<pre>Do (Mon(unit,bind)) { x <- e; f } =</pre>	<pre>bind e (fn x => f)</pre>	
<pre>Return (Mon(unit,bind)) e</pre>	<pre>= unit e</pre>	<pre>datatype Exp =</pre>

In addition the syntactic sugar of the *Do* allows a sequence of *x_i <- e_i* forms, and defines this as a nested sequence of *Do*'s. For example:

```
Do m { x1 <- e1; x2 <- e2 ; x3 <- e3 ; e4 } =
  Do m { x1 <- e1;
    Do m { x2 <- e2 ;
      Do m { x3 <- e3 ; e4 }}}}
```

Users may freely construct their own monads, though they should be very careful that their instantiation meets the monad axioms. The monad axioms, expressed in METAML's *Do* and *Return* notation are:

```
Do { x <- Return e ; z } = z[e/x]
Do { x <- m ; Return x } = m
Do { x <- Do { y <- a ; b } ; c } =
  Do { y' <- a ; Do { x <- b[y'/y] ; c } } =
  Do { y' <- a ; x <- b[y'/y] ; c }
```

5 Illustrating our compiler development method

In this section, we illustrate our method by building the front end of a compiler for a small imperative *while-language*. We proceed in three steps. First, we introduce the language and its denotational semantics by giving a monadic interpreter as a one stage METAML program. Second, we stage this interpreter by using a two stage METAML program in order to produce a compiler. Third, we illustrate the usefulness of the staging approach, by showing how using MetaML's intensional analysis tools can be used to optimize or further translate the output of a staged program.

5.1 The while-language

In this section, we introduce a simple *while-language* composed from the syntactic elements: expressions (*Exp*) and commands (*Com*). In this simple language expressions are composed of integer constants, variables, and operators. A simple algebraic datatype to describe the abstract syntax of expressions is given in METAML below:

Constant of int	(* 5 *)
Variable of string	(* x *)
Minus of (Exp * Exp)	(* x - 5 *)
Greater of (Exp * Exp)	(* x > 1 *)
Times of (Exp * Exp) ;	(* x * 4 *)

Commands include assignment, sequencing of commands, a conditional (*if* command), while loops, a print command, and a declaration which introduces new statically scoped variables. A declaration introduces a variable, provides an expression that defines its initial value, and limits its scope to the enclosing command. A simple algebraic datatype to describe the abstract syntax of commands is:

```
datatype Com =
  Assign of (string * Exp)
| Seq    of (Com * Com)
| Cond   of (Exp * Com * Com)
| While  of (Exp * Com)
| Declare of (string * Exp * Com)
| Print  of Exp;

(* ***** Example Concrete Syntax ***** *)
(* Assign   x := 1 *)
(* Seq      { x := 1; y := 2 } *)
(* Cond     if x then x := 1 else y := 1 *)
(* While    while x > 0 do x := x - 1 *)
(* Declare  declare x = 1 in x := x - 1 *)
(* Print    print x *)
```

A simple while-program in concrete syntax, such as

```
declare x = 150 in
  declare y = 200 in
    { while x > 0 do { x := x - 1; y := y - 1 };
      print y }
```

is encoded abstractly in these datatypes as follows:

```
val S1 =
  Declare("x", Constant 150,
    Declare("y", Constant 200,
      Seq(While(Greater(Variable "x", Constant 0),
        Seq(Assign("x", Minus(Variable "x",
          Constant 1)),
          Assign("y", Minus(Variable "y",
            Constant 1)))),
        Print(Variable "y"))));
```

5.2 The structure of the solution

Staging is an important technique for developing efficient programs, but it requires some forethought. To get the best results one should design algorithms with their staged solutions in mind.

The meaning of a while-program depends only on the meaning of its component expressions and commands. In the case of expressions, this meaning is a function from environments to integers. The environment is a mapping between names (which are introduced by **Declare**) and their values.

There are several ways that this mapping might be implemented. Since we intend to stage the interpreter, we break this mapping into two components. The first component, a list of names, will be completely known at compile-time. The second component, a list of integer values that behaves like a stack, will only be known at the run-time of the compiled program.

The functions that access this environment distribute their computation into two stages. First, determining at what location a name appears in the name list, and second, by accessing the correct integer from the stack at this location. In a more complicated compiler the mapping from names to locations would depend on more than just the declaration nesting depth, but the principle remains the same. Since every variable's location can be completely computed at compile-time, it is important that we do so, and that these locations appear as constants in the next stage.

Splitting the environment into two components is a standard technique (often called a binding time improvement) used by the partial evaluation community[8]. We capture this precisely by the following purely functional implementation.

```
type location = int;
type index = string list;
type stack = int list;

(* position : string -> index -> location *)
fun position name index =
  let fun pos n (nm::nms) =
        if name = nm
        then n
        else pos (n+1) nms
      in pos 1 index end;
```

```
(* fetch : location -> stack -> int *)
fun fetch n (v::vs) =
  if n = 1
  then v
  else fetch (n-1) vs;

(* put: location -> int -> stack -> stack *)
fun put n x (v::vs) =
  if n = 1
  then x::vs
  else v::(put (n-1) x vs);
```

The meaning of `Com` is a stack transformer and an output accumulator. It transforms one stack (with values of variables in scope) into another stack (with presumably different values for the same variables) while accumulating the output printed by the program.

To produce a monadic interpreter we could define a monad which encapsulates the index, the stack, and the output accumulation. Because we intend to stage the interpreter we do not encapsulate the index in the monad. We want the monad to encapsulate only the dynamic part of the environment (the stack of values where each value is accessed by its position in the stack, and the output accumulation).

The monad we use is a combination of *monad of state* and the *monad of output*.

```
datatype 'a M =
  StOut of (stack -> ('a * stack * string));
fun unStOut (StOut f) = f;
fun unit x = StOut(fn n => (x,n,""));
fun bind e f =
  StOut(fn n =>
    let val (a,n1,s1) = (unStOut e) n
        val (b,n2,s2) = unStOut(f a) n1
    in (b,n2,s1 ^ s2) end);

(* msw is the Monad of state with output *)
val msw : M Monad = Mon(unit,bind);
```

The non-standard morphisms must describe how the stack is extended (or shrunk) when new variables come into (or out of) scope; how the value of a particular variable is read or updated; and how the printed text is accumulated. Each can be thought of as an action on the stack of mutable variables, or an action on the print stream.

```
(* read : location -> int M *)
```

```
fun read i = StOut(fn ns => (fetch i ns,ns,""));

(* write : location -> int -> unit M *)
fun write i v =
  StOut(fn ns => ((), put i v ns, ""));

(* push: int -> unit M *)
fun push x = StOut(fn ns => ((), x :: ns, ""));

(* pop : unit M *)
val pop = StOut(fn (n::ns) => ((), ns, ""));

(* output: int -> unit M *)
fun output n =
  StOut(fn ns => ((), ns, (toString n) ^ " "));
```

5.3 Step 1: monadic interpreter

Because expressions do not alter the stack, or produce any output, we could give an evaluation function for expressions which is not monadic, or which uses a simpler monad than the monad defined above. We choose to use the monad of state with output throughout our implementation for two reasons. One, for simplicity of presentation, and two because if the while language semantics should evolve, using the same monad everywhere makes it easy to reuse the monadic evaluation function with few changes.

The only non-standard morphism evident in the `eval1` function is `read`, which describes how the value of a variable is obtained. The monadic interpreter for expressions takes an index mapping names to locations and returns a computation producing an integer.

```
(* eval1: Exp -> index -> int M *)
fun eval1 exp index =
  case exp of
    Constant n => Return msw n
  | Variable x => let val loc = position x index
                  in read loc end
  | Minus(x,y) =>
    Do msw { a <- eval1 x index ;
             b <- eval1 y index ;
             Return msw (a - b) }
  | Greater(x,y) =>
    Do msw { a <- eval1 x index ;
             b <- eval1 y index ;
             Return msw (if a > b
                          then 1 else 0) }
  | Times(x,y) =>
    Do msw { a <- eval1 x index ;
```

```

    b <- eval1 y index;
    Return msw0 (a * b) };

```

The interpreter for Com uses the non-standard morphisms `write`, `push`, and `pop` to transform the stack and the morphism `output` to add to the output stream.

```

(* interpret1 : Com -> index -> unit M *)
fun interpret1 stmt index =
case stmt of
  Assign(name,e) =>
    let val loc = position name index
    in Do msw0 { v <- eval1 e index ;
                write loc v } end
| Seq(s1,s2) =>
    Do msw0 { x <- interpret1 s1 index;
              y <- interpret1 s2 index;
              Return msw0 () }
| Cond(e,s1,s2) =>
    Do msw0 { x <- eval1 e index;
              if x=1
                then interpret1 s1 index
              else interpret1 s2 index }
| While(e,b) =>
    let fun loop () =
        Do msw0
          { v <- eval1 e index ;
            if v=0
              then Return msw0 ()
            else Do msw0 { interpret1 b index ;
                          loop () } }
    in loop () end
| Declare(nm,e,stmt) =>
    Do msw0 { v <- eval1 e index ;
              push v ;
              interpret1 stmt (nm::index);
              pop }
| Print e =>
    Do msw0 { v <- eval1 e index;
              output v };

```

Although `interpret1` is fairly standard, we feel that two things are worth pointing out. First, the clause for the `Declare` constructor, which calls `push` and `pop`, implicitly changes the size of the stack and explicitly changes the size of the index (`nm::index`), keeping the two in synch. It evaluates the initial value for a new variable, extends the index with the variables name, and the stack with its value, and then executes the body of the `Declare`. Afterwards it removes the binding from the stack (using `pop`), all the while implicitly threading the accumulated output. The mapping is in scope only for the body of the declaration.

Second, the clause for the `While` constructor introduces a local tail recursive function `loop`. This function emulates the body of the while. It is tempting to control the recursion introduced by the `While` by using the recursion of the `interpret1` function itself by using a clause something like:

```

| While(e,b) =>
  Do msw0
  { v <- eval1 e index ;
    if v=0
      then Return msw0 ()
    else Do msw0
      { interpret1 b index ;
        interpret1 (While(e,b)) index }
  }

```

Here, if the test of the loop is true, we run the body once (to transform the stack and accumulate output) and then repeat the whole loop again. This strategy, while correct, will have disastrous results when we stage the interpreter, as it will cause the first stage to loop infinitely.

There are two recursions going on here. First the unfolding of the finite data structure which encodes the program being compiled, and second, the recursion in the program being compiled. In an unstaged interpreter a single loop suffices. In a staged interpreter, both loops are necessary. In the first stage we only unfold the program being compiled and this must always terminate. Thus we must plan ahead as we follow our three step process. Nevertheless, despite the concessions we have made to staging, this interpreter is still clear, concise and describes the semantics of the while-language in a straight-forward manner.

5.4 Step 2: staged interpreter

To specialize the monadic interpreter to a given program we add two levels of staging annotations. The result of the first stage is the intermediate code, that if executed returns the value of the program. The use of the bracket annotation enables us to describe precisely the code that must be generated to run in the next stage. Escape annotations allow us to escape the recursive calls of the interpreter that are made when compiling a while-program.

```

(* eval2: Exp -> index -> <int M> *)

```

```

fun eval2 exp index =
case exp of
  Constant n => <Return mswow ~(lift n)>
| Variable x =>
  let val loc = position x index
  in <read ~(lift loc)> end
| Minus(x,y) =>
  <Do mswow { a <- ~(eval2 x index) ;
              b <- ~(eval2 y index);
              Return mswow (a - b) }>
| Greater(x,y) =>
  <Do mswow { a <- ~(eval2 x index) ;
              b <- ~(eval2 y index);
              Return mswow (if a '>' b
                             then 1
                             else 0) }>
| Times(x,y) =>
  <Do mswow { a <- ~(eval2 x index) ;
              b <- ~(eval2 y index);
              Return mswow (a * b) }>;

```

The `lift` operator inserts the value of `loc` as the argument to the `read` action. The value of `loc` is known in the first-stage (compile-time), so it is transformed into a constant in the second-stage (run-time) by `lift`.

To understand why the escape operators are necessary, let us consider a simple example: `eval2 (Minus(Constant 3,Constant 1)) []`. We will unfold this example by hand below:

```
eval2 (Minus(Constant 3,Constant 1)) [] =
```

```

< Do mswow
  { a <- ~(eval2 (Constant 3) []);
    b <- ~(eval2 (Constant 1) []);
    Return mswow (a-b) } >
=
< Do mswow
  { a <- ~<Return mswow 3>;
    b <- ~<Return mswow 1>;
    Return mswow (a - b) } >
=
< Do mswow
  { a <- Return mswow 3;
    b <- Return mswow 1;
    Return mswow (a - b) } >
=
< Do %mswow
  { a <- Return %mswow 3;
    b <- Return %mswow 1;
    Return %mswow (a %- b) } >

```

Each recursive call produces a bracketed piece of code which is spliced into the larger piece being con-

structed. Recall that escapes may only appear at level-1 and higher. Splicing is axiomatized by the reduction rule: $\sim\langle x \rangle \longrightarrow x$, which applies only at level-1. The final step, where `mswow` and `-` become `%mswow` and `%-`, occurs because both are free variables and are lexically captured.

Interpreter for Commands.

Staging the interpreter for commands proceeds in a similar manner:

```

(* interpret2 : Com -> index -> <unit M> *)
fun interpret2 stmt index =
case stmt of
  Assign(name,e) =>
    let val loc = position name index
    in <Do mswow { n <- ~(eval2 e index) ;
                  write ~(lift loc) n }>
    end
| Seq(s1,s2) =>
  <Do mswow { x <- ~(interpret2 s1 index);
              y <- ~(interpret2 s2 index);
              Return mswow () }>
| Cond(e,s1,s2) =>
  <Do mswow
    { x <- ~(eval2 e index);
      if x=1
      then ~(interpret2 s1 index)
      else ~(interpret2 s2 index)}>
| While(e,b) =>
  <let fun loop () =
        Do mswow
          { v <- ~(eval2 e index);
            if v=0
            then Return mswow ()
            else Do mswow
                  { q <- ~(interpret2 b index);
                    loop () }
          }
        in loop () end>
| Declare(nm,e,stmt) =>
  <Do mswow { x <- ~(eval2 e index) ;
              push x ;
              ~(interpret2 stmt (nm::index)) ;
              pop }>
| Print e =>
  <Do mswow { x <- ~(eval2 e index) ;
              output x }>;

```

5.4.1 An example.

The function `interpret2` generates a piece of code from a `Com` datatype. To illustrate this we apply it to the simple program: `declare x = 10 in { x := x - 1; print x }` and obtain:

```
<Do %mswo
{ a <- Return %mswo 10
; %push a
; Do %mswo
  { e <- Do %mswo
    { d <- Do %mswo
      { b <- %read 1
      ; c <- Return %mswo 1
      ; Return %mswo b %- c
      }
    ; %write 1 d
    }
  ; g <- Do %mswo
    { f <- %read 1
    ; %output f
    }
  ; Return %mswo ()
  }
; %pop
}>
```

Note that the staged program is essentially a compiler, translating the syntactic representation of the while-program into the above monadic object-program that will compute its meaning. Note that in the object-program all of the compile-time operations have disappeared. This object-program is fully executable. Simply by using the `run` operator of METAML, it can be executed for prototyping purposes.

6 Step 3: Back-end translation and intermediate code optimization

METAML is a meta-programming system. It has an object language and a meta-language. Meta-programs are programs that manipulate object programs. In METAML both the object language and the meta-language are ML. In METAML an object-program is both a data structure that can be manipulated, and a program that can be run.

This duality plays an important role in target code generation. The result of applying the staged inter-

preter from the previous step (a meta-program) to a DSL program to be compiled is a highly constrained residual program (an object program). This program is both a data-structure and a program, so it can be both directly executed (rapid prototype) and analyzed.

We use the object-code analysis capabilities of MetaML to transform the object program into the final target language. This analysis can include both source to source transformations, or translation into another form (i.e. intermediate code, assembly language, or target language).

Control over the form of the residual program is crucial here. The residual program is always an ML program (ML is the object language). But the user can control the form of this ML program. A goal of the translation is to make the object program use only those ML features directly supported by the target language. For example, we may structure the staged interpreter such that the residual program is first order, or just a sequence of primitive actions encoded as non-standard morphisms in the monad. This is where we connect the abstract monadic actions to their efficient implementations.

The object program produced above is an ML code fragment. It can be executed or analyzed. The code produced by `interpret2` is a restricted subset of ML. Disregarding the higher-order functions implicit in the monad, it is first order, and contains only `Do` expressions, `Return` expressions, `if` expressions, calls to the non-standard morphisms `read`, `write`, `push`, `pop`, and `output`, primitive arithmetic operators `-` and `'>'`, and local looping functions (like `loop` above). The code is so regular that it can be captured by a simple grammar. The next step is to analyze this code to make the final translation to the target language, or to apply some ML-source to ML-source level optimizations. The reader might notice that the object-program above could be considerably, further simplified by applying the monad laws. There are many opportunities for doing so. After these laws are applied we obtain the much more satisfying:

```
<Do %mswo
{ %push 10
; a <- %read 1
; b <- Return %mswo a %- 1
; c <- %write 1 b
; d <- %read 1
; e <- %output d
```

```

; Return %mswo ()
; %pop
}>

```

In addition to the monad laws which hold for all monads, we can also use laws which hold for particular non-standard morphisms. For instance, in the example above, we could avoid the second read of location 1 using the following rule:

```

Do { e1
  ; c <- %write 1 b
  ; d <- %read 1; e2
}
=
Do { e
  ; c <- %write 1 b
  ; e2[b/d]
}

```

Every target language will have many such laws, and because our target language is both executable-code, and data-structure we can perform these optimizations. The final step is to translate the ML code fragment into the target language. This step uses the same intensional analysis of code capabilities of the optimization steps, and is the subject of the next section.

6.1 Intensional analysis of code fragments

In this section, we outline how we do intensional analysis of residual code. We provide a high-level pattern matching based interface. Code patterns can be constructed by placing brackets around code. For example a pattern that matches the literal 5 can be constructed by:

```

-| fun is5 <5> = true
  | is5 _ = false;
val is5 = fn : <int> -> bool

-| is5 (lift (1+4));
val it = true : bool

-| is5 <0>;
val it = false : bool

```

The function `is5` matches its argument to the constant pattern `<5>` if it succeeds it returns `true` else

`false`. Pattern variables in code patterns are indicated by escaping variables in the code pattern.

```

-| fun parts < ~x + ~y > = SOME(x,y)
  | parts _ = NONE;
val parts = fn : <int> -> (<int> * <int>) option

-| parts <6 + 7>;
val it = SOME (<6>,<7>) : (<int> * <int>) option

-| parts <2>;
val it = NONE : (<int> * <int>) option

```

The function `parts` matches its argument against the pattern `< ~x + ~y >`. If its argument is a piece of code which is the sum of two sub terms, it binds the pattern variable `x` to the left subterm and the pattern variable `y` to the right subterm.

We use higher-order pattern variables[22, 21] for code patterns that contain binding occurrences, such as lambda expressions, let expressions, do expressions, or functions.

For example, a high-order pattern that matches the code of a function `<fn x => ...>`, of type `<'a -> 'b>` is written in eta-expanded form `<fn x => ~(g <x>>>`. When the pattern matches, the matching binds the higher-order pattern variable `g` to a function with type `<'a> -> <'b>`

Every higher order pattern variable must be in fully saturated form, by applying it to all the bound variables of the code pattern. For example if `g` is a higher-order pattern variable with type `<'a> -> <'b> -> <'c>` then we must write `~(g <x> <y>>`. The arguments to the higher-order pattern variable must be explicit bracketed variables, one for each variable bound in the code pattern at the context where the higher-order pattern appears. A higher-order pattern variable is used like a function on the right-hand side of a matching construct.

For example functions which implement the three monad axioms are written as follows:

```

fun monad1
  <do msw
    { x <- return msw ~e
      ; ~(z <x>) }>
= z e

fun monad2 <do msw { x <- ~m; return x }> = m

```

```

fun monad3
  <do msw0
    { x <- do msw0 {y <- ~a
      ; ~(b <y>)}
      ; ~(c <x> )}>
= <do msw0 { y' <- ~a
  ; do msw0 { z <- ~(b <y'>)
    ; ~(c <z> )}> }>

```

When the function `monad1` is applied to the code `<do msw0 {a <- return msw0 (g 3); h(a + 2)}>`, the pattern variable `e` is bound to the function `fn x => <h(~x + 2)>` which has the type `<int> -> <int M>`. The right-hand side of `monad1` rebuilds a new code fragment, substituting formal parameter `x` of `e` by `<g 3>`, constructing the code `<h((g 3)+ 2)>`.

This technique can be used to build optimizations, or to translate a residual program into a target language.

7 Conclusion

The important issues of efficient language implementation by refinement from high-level specifications are: the efficient use of the underlying target environment, and removing the layer of interpretative computation introduced by such specifications. We have shown that monads and staging are the right abstraction mechanisms to accomplish the task. To effectively use these tools we propose that DSL implementers follow a well defined method. We reiterate our method here:

- **Domain analysis.** The problem domain is analyzed to find the common abstractions around which the language is designed. This step is perhaps the most important step in a good language design. It has been studied extensively by others [32, 2, 3]. Our research group has been investigating the integration of DSL design and domain analysis for several years. Recently Widen and Hook have summarized a “top level” view of this integration, which is called the Software Design Automation (SDA) method [33]. This method provides a design process and many synthesis techniques to facilitate the integration of traditional domain

analysis activities with language design and implementation. The method we propose can be used in the context of SDA. It specifically addresses the language implementation phase of the process.

- **Definitional interpreter.** Once the language has been identified, the next step is to provide it with a semantics given as a pure functional interpreter. This program can be thought of as its high-level definition [14, 25]. high-level interpreters are usually easy to construct and provide a reference which can be consulted to resolve any ambiguity in the language specification discovered in further steps. By building it in an executable framework (a functional language, such as Haskell or ML) it also provides a rapid prototype against which expectations can be measured.
- **Binding time improvements.** The next step requires a binding separation [8]. By identifying compile-time versus run-time data structures in the definitional interpreter, we can separate those with *both* components into separate data-structures. Examples of binding time improvements include the separation of environments, which map names to values, into a compile-time index and a run-time stack, and the introduction of a local recursive function to separate the recursion which drives the analysis of the syntax of the program being interpreted from the recursion that encodes the looping of the `while` command.
- **Target domain analysis.** The next step is to analyze the target language to identify the primitive implementation features that will support the translation. This step is usually straight-forward as the target language is often fixed, and well understood.

- **Design a monad.** The next step is to design a monad to capture the effects and actions implicit in the target language. This is a hard step in the process since it requires both abstract knowledge about the structure and properties of monads, and detailed concrete knowledge about the target domain. The choices made in this step influence the structure of the monad, the structure of the monadic interpreter, and the run-time system which interacts with the low-level effects of the target language.

Once the monad is designed, an implementation for the monad as a pure functional emulation must be produced. The implementation

must emulate the actions in a purely functional setting by explicitly threading abstract representations of the actions such as “stores”, “I/O streams”, or “exception continuations” in and out of all computations.

- **Monadic Interpreter.** The next step is to refine the purely functional definitional interpreter into one written in a monadic style [28, 24, 13]. This implementation is still purely functional because the actions of the monad are emulated in a functional style. But because the actions are now explicit, we have moved the form of definition closer to the target language. This step often requires a big change to the structure of the source code, because the monad makes implicit much of the “plumbing” explicit in the interpreter. The cost of this restructuring is not without benefit. The removal of the explicit plumbing results in programs which are simpler, and more immune to future changes.
- **Staging.** The next step completes the binding-time separation begun in the binding time improvement step. That step separated the compile-time *data* from the run-time data. Staging separates the compile-time *computations* from the run-time computations. This is done by placing explicit staging annotations in the program written in METAML. Staging is the crucial step that differentiates an (inefficient) interpreter from an (efficient) compiler.
- **Transformation of residual code.**

The residual object-program produced by a staged interpreter is both a data structure that can be manipulated, and a program that can be run. Control over the form of the residual program is crucial here. The residual program is always an ML program (ML is the object language). But the user can control the form of this ML program. A goal of the translation is to make the object program use only those ML features directly supported by the target language. The restricted form of the residual object program make it possible to use the intensional analysis of object-code tools provided by MetaML to easily build the final translation step to the target language.

7.1 Benefits of the approach

This paper illustrated a step by step method for constructing correct and efficient implementations of DSLs. The method has the following advantages over building a DSL implementation in an ad-hoc fashion.

- **Simplicity.** We divide the task of DSL implementation of DSL into small manageable tasks. The compiler is constructed by a method of refinement, and we use special abstraction mechanisms so that each step addresses only a single aspect of the compiler.
- **Reuse.** Our method provides many opportunities for reuse. By using the abstraction methods of monads and staging, much of the code remains unchanged between refinement steps. In addition, monad implementations are reusable across DSLs, and multiple DLS using the same target language can reuse the intensional analysis.
- **Control.** Instead of using a fixed set of techniques or tool to generate compilers, we outline a method which provides users control over each step. A good impedance match between low-level features of the target language and the high-level DSL is necessary for good performance. Since every compiler is different, users need such fine grained control.
- **Correctness.** The METAML type system provides major support for ensuring the correctness of the compilers generated. It is simply not possible to write a type-incorrect translation. But type-correctness is not enough. We wish to prove other correctness properties as well, such as the equivalence between the artifacts produced by each step of the method. We believe that it is possible for each step to make explicit its proof obligations, and because each step produces a functional program, it is possible to use equational reasoning to prove these obligations

7.2 The Implementation

Everything you have seen in this paper, except the higher order pattern matching over code, has been

implemented in the METAML implementation. The examples are actual runs of the system.

The higher order pattern matching is currently under development. We found the normalizing effect of the monad laws so compelling that we implemented them in an ad-hoc fashion inside the METAML system.

8 Acknowledgments

The authors would like to acknowledge generous support from the USAF Air Materiel Command, contract #F19628-96-C-0161; the National Science Foundation, grants #IRI-9625462 and #CCR-9803880; and the Department of Defense.

References

- [1] Anders Bondorf and Jens Palsberg. Compiling actions by partial evaluation. In *Conference on Functional Programming Languages and Computer Architecture*, pages 308–320, New York, June 1993. ACM Press. Copenhagen.
- [2] Grady Campbell. Abstraction-based reuse repositories. Technical Report REUSE-REPOSITORIES-89041-N, Software Productivity Consortium Services Corporation, 2214 Rock Hill Road, Herndon, Virginia 22070, June 1989.
- [3] Grady Campbell, Stuart Faulk, and David Weiss. Introduction to Synthesis. Technical Report INTRO-SYNTHESIS-PROCESS-90019-N, Software Productivity Consortium Services Corporation, 2214 Rock Hill Road, Herndon, Virginia 22070, 1990.
- [4] Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg Beach, Florida, 21–24 January 1996.
- [5] O Danvy, J Koslowski, and K Malmkjær. Compiling monads. Technical Report CIS-92-3, Kansas State University, Manhattan, Kansas, December 91.
- [6] Robert Glück and Jesper Jørgensen. Efficient multi-level generating extensions for program specialization. In S. D. Swierstra and M. Hermenegildo, editors, *Programming Languages: Implementations, Logics and Programs (PLILP'95)*, volume 982 of *Lecture Notes in Computer Science*, pages 259–278. Springer-Verlag, 1995.
- [7] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *In FPCA '93: Conference on Functional Programming Languages and Computer Architecture*, pages 52–64. ACM Press, 1993. (Appears, in extended form, in the *Journal of Functional Programming*, 5, 1, Cambridge University Press, January 1995.).
- [8] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Series editor C. A. R. Hoare. Prentice Hall International, International Series in Computer Science, June 1993. ISBN number 0-13-020249-5 (pbk).
- [9] Paul Hudak Simon Peyton Jones, Philip Wadler, Brian Boutel, John Fairbairn, Joseph Fasel, Maria M. Guzman, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell. *SIGPLAN Notices*, 27(5):Section R, 1992.
- [10] Peter Lee. *Realistic Compiler Generation*. Foundations of Computing Series. MIT Press, 1989.
- [11] Mark Leone and Peter Lee. A declarative approach to run-time code generation. In *Workshop on Compiler Support for System Software (WCSSS)*, February 1996.
- [12] Sheng Liang and Paul Hudak. Modular denotational semantics for compiler construction. In *ESOP'96: 6th European Symposium on Programming*, number 1058 in LNCS, pages 333–343, Linköping, Sweden, January 1996.
- [13] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *ACM Symposium on Principles of Programming Languages*, pages 333–343, San Francisco, California, January 1995.
- [14] P. Mosses. Denotational semantics. In J. van Leeuwen, editor, *Handbook of Theoretical Com-*

- puter Science. Elsevier Science Publishers B. V. (North-Holland), 1990.
- [15] Peter D. Mosses. SIS-semantics implementation system, reference manual and users guide. Technical Report DAIMI report MD-30, University of Aarhus, Aarhus, Denmark, 1979.
- [16] Peter D. Mosses. Action semantics. *Cambridge Tracts in Theoretical Computer Science*, (26), 1992.
- [17] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *23rd ACM Symposium on Principles of Programming Languages*, St. Petersburg, Florida, January 1996.
- [18] L. Paulson. *Methods and Tools for Compiler Construction*, B. Lorho (editor). Cambridge University Press, 1984.
- [19] Lawrence Paulson. A semantics directed compiler generator. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 224–233. ACM, January 1982.
- [20] John Peterson, Kevin Hammond, et al. Report on the programming language haskell, a non-strict purely-functional programming language, version 1.3. Technical report, Yale University, May 1996.
- [21] Frank Pfenning, Jolle Despeyroux, and Carsten Schrmann. Primitive recursion for higher-order abstract syntax. In *Third International Conference on Typed Lambda Calculi and Applications (TLCA'97)*, pages 147–163, Nancy, France, April 1997.
- [22] Frank Pfenning, Gilles Dowek, Threse Hardin, and Claude Kirchner. Unification via explicit substitutions: The case of higher-order patterns. In *Joint International Conference and Symposium on Logic Programming (JICSLP'96)*, Bonn, Germany, September 1996.
- [23] Calton Pu and Jonathan Walpole. A study of dynamic optimization techniques: Lessons and directions in kernel design. Technical Report OGI-CSE-93-007, Oregon Graduate Institute of Science and Technology, 1993.
- [24] Guy Steele. Building interpreters by composing monads. In *21st Annual ACM Symposium on Principles of Programming Languages (POPL'94)*, Portland, Oregon, January 1994.
- [25] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, Cambridge, Massachusetts, 1977.
- [26] Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. Multi-stage programming: Axiomatization and type-safety. In *25th International Colloquium on Automata, Languages, and Programming*, Aalborg, Denmark, 13–17 July 1998.
- [27] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantic based program manipulations PEPM'97*, Amsterdam, pages 203–217. ACM, 1997.
- [28] Philip Wadler. Comprehending monads. *Proceedings of the ACM Symposium on Lisp and Functional Programming*, Nice, France, pages 61–78, June 1990.
- [29] Philip Wadler. Comprehending monads. *Proceedings of the ACM Symposium on Lisp and Functional Programming*, Nice, France, pages 61–78, June 1990.
- [30] Philip Wadler. The essence of functional programming (invited talk). In *19th ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, January 1992.
- [31] Philip Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of LNCS. Springer Verlag, 1995.
- [32] Tanya Widen. Formal language design in the context of domain engineering. Master's thesis, Department of Computer Science and Engineering, Oregon Graduate Institute, October 1997.
- [33] Tanya Widen and James Hook. Software design automation: Language design in the context of domain engineering. In *The 10th International Conference on Software Engineering & Knowledge Engineering (SEKE'98)*, pages 308–317, San Francisco Bay, California, June 1998.

Monadic Robotics

John Peterson

Yale University

peterson-john@cs.yale.edu, <http://www.cs.yale.edu/homes/peterson-john.html>

Greg Hager

The Johns Hopkins University

hager@cs.jhu.edu, <http://www.cs.jhu.edu.dom/~hager>

Abstract

We have developed a domain specific language for the construction of robot controllers, Frob (Functional ROBotics). The semantic basis for Frob is Functional Reactive Programming, or simply FRP, a purely functional model of continuous time, interactive systems. FRP is built around two basic abstractions: behaviors, values defined continuously in time, and events, discrete occurrences in time. On this foundation, we have constructed abstractions specific to the domain of robotics. Frob adds yet another abstraction: the *task*, a basic unit of work defined by a continuous behavior and a terminating event.

This paper examines two interrelated aspects of Frob. First, we study the design of systems based on FRP and how abstractions defined using FRP can capture essential domain-specific concepts for systems involving interaction over time. Second, we demonstrate an application of *monads*, used here to implement Frob tasks. By placing our task abstraction in a monadic framework, we are able to organize task semantics in a modular way, allowing new capabilities to be added without pervasive changes to the system.

We present several robot control algorithms specified using Frob. These programs are clear, succinct, and modular, demonstrating the power of our approach.

1 Introduction

A successful DSL combines the vocabulary (values and primitive operations) of an underlying domain with abstractions that capture useful patterns in the vocabulary. Ideally, these abstractions organize the vocabulary into structures that support clarity and modularity in the domain of interest. In robotic control, this basic vocabulary is quite simple: it consists of feedback systems connecting the robot sensors and effectors. The more difficult task is to build complex behaviors by sequencing among various control disciplines, guided by overall plans and objectives. Controllers must be robust and effective, capable of complex interactions with an uncertain environment. While basic feedback systems are well understood, constructing controllers remains a serious software engineering challenge. Many different high-level architectures have been proposed but no one methodology addresses all problems, making this an ideal area for the application of DSL technology.

Frob is an embedded DSL for robotic control systems. Frob is built on top of Functional Reactive Programming (FRP), which is in turn built on top of Haskell, a lazy, purely functional programming language[14]. Frob hides the details of low-level robot operations and promotes a style of programming largely independent of the underlying hardware. It also promotes a *declarative* style of specification: one that is not cluttered by low level implementation details.

An advantage of Frob (as well as many DSLs) is that it is *architecture neutral*. That is, instead of defining a specific system architecture (organization or basic design pattern) it enables arbitrary architectures to be defined in a high level, reusable manner. As

an embedded DSL, Frob includes the capabilities of a fully-featured functional programming language, Haskell.

This paper addresses both the Frob language itself, its capabilities, usage, and effectiveness, and the implementation of Frob. In particular, we examine the use of a *monad* to implement one of the essential semantic components of Frob. We demonstrate how “off the shelf” monadic constructs may be incorporated into a domain specific language to express its semantic foundation clearly and, more importantly, in a modular manner. We address monads from a practical vantage rather than a theoretical one, emphasizing their usage and benefits within our domain.

This paper contains many examples written in Haskell. Readers must have at least a passing familiarity with the syntax, primitives, and types of Haskell. Those unfamiliar with Haskell will find www.haskell.org has much helpful information. Although we make extended use of Functional Reactive Programming, we attempt to explain FRP constructs as they are used. No prior understanding of monads is required.

The remainder of this paper is organized as follows. Section 2 discusses the domain of robot control and essentials of FRP and monads. Section 3 demonstrates the construction of the task monad in an incremental manner, adding features one by one and examining the impact on the system as the definition of a task changes. In Section 4, a number of non-trivial examples of Frob programming are presented. Section 5 concludes.

2 Background

2.1 The Problem Domain

Programming robots operating in the real world provides a unique example of a programming system operating in conjunction with the physical world. As such, any system must coordinate multiple ongoing control processes, detect special events governing task execution, and supply data structures and language primitives appropriate to the domain.

Unsurprisingly, the development of robot program-

ming systems has been an area of active research in robotics (see [2] for a recent collection of articles in this area). Many of these languages are realized by defining data structures and certain specialized library routines to existing languages (notably C [6, 5, 10, 16], Lisp [1], Pascal [11], and Basic [15]). In particular, most of these “languages” include special functions or commands that operate in the time domain. For example, VAL includes a command to move the robot to a given spatial location. This command operates asynchronously and has, thereby, the side-effect of “stitching together” multiple motions if they are supplied in rapid order. Likewise, other embedded languages such as the “Behavior Language” of Brooks [1] and the Reactive Control Framework of Khosla [16] provide rich programming environments for the coordination of time-domain processes. AML [17] is an exceptional case — it is a language designed from scratch. As such, it supplies similar capabilities to VAL, but with the addition of enhanced error handling capabilities for robot program execution.

Despite the proliferation of robot programming systems, relatively little work has been done on deriving a principled basis for them. For example, none of the languages cited above has a formal (or in many cases even informal!) semantics. Counter to this trend, Lyons [9] provides a compositional paradigm for expressing robot plans (collections of atomic actions) that are sequenced and coordinated using a rich set of primitives. The notion of continuous behavior and discrete event-based transitions is also introduced. However, no transparent implementation of the language (or even realistic examples) is presented.

In contrast, Frob has a transparent, extensible, and semantically clear basis, and is a practical, useful tool for implementing robot programming systems. While Frob is also a library embedded in an existing language, the abstractions defining Frob are at a higher level than those used to embed controllers in languages such as C. While a controller embedded in C or C++ could, in theory, handle the same sort of abstractions we have used in Frob, their implementation would be much more difficult and the reliability of the system would suffer in the absence of a good polymorphic type system.

2.2 About Haskell

Since Frob inherits the syntax, type system, and libraries of Haskell users of Frob must, of necessity, learn about Haskell. This section contains a brief overview of the Haskell features used in Frob; those familiar with Haskell may wish to skip to the next section.

Basic Haskell features include the following:

- Variables: these start with lower case letters, such as `x` or `robotPosition`. Variables may include `_` and `'` characters, so `x'` is also a variable name. Operators are composed of punctuation, such as `=>` or `.|..`.
- Function application: Haskell does not use `f(x,y)` style notation for function calling. Instead, the parenthesis and commas are omitted, as in `f x y`. Parentheses may be used for grouping, as in `f (g x) (h y)`, which would be written `f(g(x),h(y))` in other languages.
- Infix operators: examples include `x + y` or `e ==> f`. An infix operation is converted into an ordinary variable using parentheses, as in `(==>)` or `(+)`. Ordinary functions, such as `f`, can be used in the infix style when surrounded by backquotes, as in `x `f` y`, which is the same as `f x y`. Application takes precedence over infix operators, so `f x+y z` parses as `(f x)+(y z)`.
- Layout: indentation separates definitions: each definition in a list must start in the same column and the list is terminated by anything in a preceding column. For example, in

```
let x = 1
    y = 2
    in x+2
```

the indentation of the `y` must exactly match that of `x`.

- Definitions: the `=` in Haskell creates a definition. You can define a constant, as in `x = 3`, or a function, as in `f x y = x + y`. As with function application, no parentheses are needed around the function parameters.
- Lambda abstractions: functions need not be named. The expression `\x y -> x + y`

is an anonymous function. For example, you can pass a function as a parameter: `f (\y -> y + 2)`. There's no difference between `f x y = x + y` and `f = \x y -> x + y`.

- Type signatures: the polymorphic types in Haskell are quite descriptive. Types are inferred, allowing type signatures to be omitted, but for clarity we include signatures in all examples. Type signatures supply valuable documentation and make type errors easier to diagnose. The syntax of a signature declaration is `x :: Int`, where this defines the type of `x` to be `Int`. Type signatures are generally placed immediately preceding the associated definition.
- Function types: the type of a function from type `t1` to type `t2` is written `t1 -> t2`. A function with more than one argument will have more than one arrow in its type. Watch for parenthesis, though: the type `f :: (Int -> Int) -> Int` defines a function with one argument, a function from `Int` to `Int`, rather than two `Int` arguments; that would be `f :: Int -> Int -> Int`.
- Currying: you don't need to pass all of the arguments to a function at once. A call to `f :: Int -> Int -> Int` without the second argument, as in `f 3`, results in a function that takes the remaining argument.
- Polymorphic types: lower case identifiers in type expressions are type variables. These scope over a single type signature and denote type equality. Many types are parameterized; the parameters are passed using the same syntax that expressions use. For example, the signature

```
(==>) :: Event a -> (a -> b) -> Event b
```

defines an operator, `==>`, that takes an `Event` parameterized over some type `a` and a function from type `a` to another type `b`, yielding an `Event` parameterized over type `b`. Polymorphic types are an essential part of Frob; many built-in Frob operators are may be almost completely described by their type signature.

- Contexts: In Haskell, overloading is manifested in type signatures that contain a context: a set of constraints on type variables, prefixing an ordinary signature. For example, the type

```
(>) :: Ord a => a -> a -> Bool
```

indicates that two arguments to `>` are of the same type and that this type must be in class `Ord`. A Haskell type class is similar to a Java interface.

- **Tuples and Lists:** A tuple is a simple way of grouping two or more values. Tuples use parentheses and commas: `(x,y)` is an expression that combines `x` and `y` into a single tuple. The elements of a tuple may have different types. Lists are written using square brackets: `[1,2,3]` is a list of three integers. The `:` operator adds a new element to the front of a list; `[1,2,3] = 1:2:3:[]`. Many list functions are pre-defined in Haskell.
- **The Maybe type:** the type `Maybe a` denotes either a value of type `a`, `Just x`, or `Nothing`. `Maybe` is often used where C programmers would use a pointer that may be void.
- **The void type:** the type `()`, pronounced "void", is used in situations where no value is needed. The only value of this type is `()`.

2.3 Functional Reactive Programming

In developing Frob we have relied on our experience working with *Fran*, a DSL embedded in Haskell for functional reactive animation [4, 3]. Functional Reactive Programming can be thought of as *Fran* without animation: the basic events, behaviors, and reactivity without operations specific to graphics.

At the core of FRP is the notion of *dynamically evolving* values. A variable in FRP may denote an ongoing process in time rather than a static value. There are two sorts of evolving values: continuous behaviors and discrete events. A behavior is defined for all time values while events have a value only at some discrete set of times.

For a type `t`, the type `Behavior t` is an evolving quantity of type `t`. Behaviors are used to model continuous values: a value of type `Behavior SonarReading` represents the values taken from the sonars; `Behavior Point2` represents the position of the robot. Expressions in the behavioral domain are not significantly different from static expressions. Through overloading, most static operators operate also in the dynamic world: users see little difference between programming with static values and with behaviors. For example, the following declaration is typical of Frob:

```
error :: Robot -> Behavior Float
error r =
  limit
    (velocity r * sin thetamax)
    (setpoint - leftSonar r)
  where limit m v = (-m) 'max' v 'min' m
```

This example shows a function mapping robot sensors (as selected by the `velocity` and `leftSonar` functions) onto a time-varying float, part of a larger control system. The details of this example are unimportant; the point is that writing functions over behaviors is little different from writing functions for static value.

Behaviors hide the underlying details of clocking and sampling, presenting an illusion of continuous time. Behaviors also support operators not found in the static world: both `integral` and `derivative`, for example, exploit time flow. As a further example of the expressive power of behaviors, consider the following:

```
atMin :: Ord a =>
  Behavior a -> Behavior b -> Behavior b
```

This returns the value of the second behavior at the time the first behavior is at its minimum.

The other essential abstraction supplied with FRP is the event. The type `Event t` denotes a process that generates discreet values (messages) of type `t` at specific instances. Some components of the system are best represented by events rather than behaviors. For example, the bumpers are of type `Event BumperEvent`; occurrences happen when one of the robot bumper switches is activated. The console keyboard has type `Event Char`, where each key-press generates an event occurrence.

Events may be synthesized from boolean behaviors, using the `predicate` function:

```
predicate :: Behavior Bool -> Event ()
```

This can be thought of monitoring a boolean behavior and sending a message when it becomes true. This definition uses `predicate` to generate an event when an underlying condition first holds:

```
stopit :: Robot -> Event ()
stopit r = predicate
  (time > timeMax || frontSonarB r < 20)
```

This event occurs when either the current time passes some maximum value or when an object appears less than 20 cm away on the front sonar of the robot.

Within FRP, a robot controller is simply a function from the robot sensors, represented using behaviors and events bundled into the `Robot` type, onto its effectors, behaviors and events that drive the wheels and any other systems controlled by the robot. The flow of time is hidden with the FRP abstractions; the user sees a purely functional mapping from inputs to outputs.

Is this all we need? Perhaps; complex robot controllers can be constructed using only the basic FRP primitives. However, such controllers have a number of problems:

- While FRP is well suited for the low-level control systems present in this domain, it lacks higher level constructs needed to plainly express robot behaviors at a high level.
- Controllers may be complicated by “plumbing” code needed to propagate values in a functional manner.
- Hard to understand FRP constructs are sometimes required. While users easily comprehend basic event and behavioral operators, FRP is also littered with arcane (but essential) operators such as `snapshot`, `switcher`, or `withElem` that are unfamiliar to users and are not a natural part to the underlying domain.

Our goal is to create better abstractions: ones that embody patterns that are familiar to domain engineers and that have well-defined semantic properties.

2.4 Monads as a Modular Abstraction Tool

Monads have been surrounded by a great deal of hype in the functional programming community. This has led those outside of this community to ask questions such as “What the heck are monads anyway?”, “If monads are so useful why don’t they have them in Java?”, and “Do I have to understand category theory to write Haskell programs?”. In this

section we will attempt to demystify monads somewhat.

First, why don’t C programmers or Java programmers use monads? The answer is really quite simple: monads don’t actually do anything new. Monads are used for state, exceptions, backtracking, and many other things that programmers have long done without monads. What the monad does is allow many well-understood constructs to be explained conveniently in purely functional terms. Outside a purely functional language, it’s usually easier (but maybe not better) to do what you want directly without involving monads.

In a pure language, though, monads are an ideal way to capture the essential semantics of a domain without compromising purity or modularity. Monads hide the “gears and wheels” of the domain from the user while also presenting a simple, intuitive interface to the user. The user (as opposed to the DSL designer) sees only sequencing and the return operator, together with ‘magic’ functions that reach inside to these gears and wheels in some way. Monadic programming is made more readable in Haskell using the `do` notation. Users of the DSL don’t really have to know anything about monads at all; they simply “wire up” their program using `do` and the rest of the monadic internals are unseen.

The most important feature of the monadic approach is modularity: new features may be added without breaking existing code. Under the hood, though, interactions between different features in a monad are out in the open. This is the advantage of a purely functional style: the interplay among these various features is very explicit.

Another advantage of monads is that there are many “off the shelf” monadic constructions available. There is no need to re-invent a basic semantic building block such as exception handling; this is already well understood. A DSL designer may combine many such building blocks into a very domain-specific monad. There are also a number of algebraic properties that make monadic programs easier to understand and reason about.

Going back to a very concrete level, a monad in Haskell defined with an instance declaration that associates a type with the two monadic operations in the class `Monad`:

```
class Monad m where
```

```

(>>=)    :: m a -> (a -> m b) -> m b
return   :: a -> m a

```

The operators are simple enough: `>>=` (or *bind*) is sequential composition while `return` defines an “empty” computation. A special syntax, `do` notation, makes calls to `>>=` more readable. In addition to this instance declaration, other functions may reach inside the monad, hooking into its internals. For example, consider the state monad. Here, the `bind` and `return` define the propagation of the state from computation to computation. To actually reach inside to the state, additional functions must be written to get at this internal state. The following example shows the declarations needed to define a monad over a container type `T` that, internally, maintains a state of type `S`:

```

data S = ...

-- A computation in T that returns type a
-- is a function that takes a state and returns
-- an updated state and an a.

data T a = T (\S -> (S, a))

instance Monad T where
  (T f1) >>= c2 =
    T (\state ->
      let (state', r) = f1 state
      in T f2 = c2 r in
      f2 state')
  return k = T (\state -> (state, k))

getState :: T S
getState = T (\state -> (state, state))

setState :: S -> T ()
setState state = T (\_ -> (state, ()))

runT :: S -> T a -> (S, a)
runT state (T f) = f state

```

The `Monad` instance explicates the passing of the state from the first computation to the second. The `getState` and `setState` reach inside this particular monad to access the normally hidden state. Finally, the `runT` function runs a computation in monad `T`, passing in an initial state and producing the final state and returned value.

We may add new capabilities to the state monad (exception handling, for example) without changing any of the user level code that uses the monad. That is, we may often enrich a monad’s vocabulary without altering “sentences” expressed in the old

vocabulary.

Perhaps the best introduction to the practical use of monads is Wadler’s “The Essence of Functional Programming”[18].

3 Implementing Frob

The basic implementation of Frob is discussed in [13] and [12]. Here, we examine only tasks and their use of monads.

3.1 The Basic Task Monad

Rather than present the full definition of Frob tasks up front, we will instead develop the task abstraction incrementally, adding features one by one and showing how each incremental extension in the expressiveness of tasks affects programs and the task implementation. Our purpose here is twofold: to show the ability of functional reactive programming to define the abstractions needed for our domain and, more importantly, to show how the use of a monad to organize the task structure promotes modularity.

The essential idea behind a task is quite simple: the type `Task a b` defines a behavior (actually, any reactive value), `a`, over some duration and then exits with a value of type `b`. In terms of FRP, a task is represented as

```
behavior 'untilB' event ==> nextTask
```

where `untilB` switches the behavior upon occurrence of the terminating event. The `==>` operator passes the value generated by the event to the next task. Tasks are a natural abstraction in this domain: they couple a continuous control system (a behavior) with an event that moves the system to a new mode of operation. Tasks are not restricted to the top level of the system: any reactive value (event or behavior) may be defined as a task; many tasks may be active at one time.

Initially, the task monad requires only one instrument from the monadic toolbox: a continuation to carry the computation to the next task. This is

implemented by a type, `Task`, and an instance declaration for the standard Haskell `Monad` class:

```
data Task a b = Task ((b -> a) -> a)
unTask (Task t) = t

-- standard continuation monad
instance Monad Task where
  (Task f) >>= g =
    Task (\c -> f (\r -> unTask (g r) c))
  return k =
    Task (\c -> c k)
```

This defines a structure for combining computations (the glue); we still need to define the computations themselves. Here is a simple task creation function:

```
mkTask :: (Behavior a, Event b) -> Task a b
mkTask (b,e) =
  Task (\c -> b 'untilB' e ==> c)
```

Now that we can create tasks and sequence tasks, how do we get out of the `Task` world? After all, the robot controller is defined in terms of behaviors, not tasks. That is, we need to convert a task into a behavior. This brings up a small problem: what to do when the task completes? That is, what is the initial value of the continuation argument? One way out of this dilemma is to pass in an additional behavior to take control after the task exits:

```
runTask :: Task a b -> a -> a
runTask (Task t) finalB = t (const finalB)
```

We now have everything needed to write a simple robot controller. Here are two simple tasks:

```
goAhead, turnRight, runAround ::
  Robot -> Task WheelControlB ()
goAhead r =
  mkTask (pairB 10 0)
    (predicate (frontSonar r < 20))
turnRight r =
  mkTask (pairB 0 0.5)
    (predicate (frontSonar r > 30))
runAround r = do goAhead r
  turnRight r
  runAround r -- loop forever
main =
  runController
    (\r -> runTask (runAround r) undefined)
```

The wheel controls are defined by a pair of numbers, constructed using `pairB`, with the first being the forward velocity and the second the turn

rate. The `runController` function executes a controller, a function from sensors (`Robot`) to effectors (`WheelControlB`). Since `runAround` is not a terminating task there is no reason to pass a final behavior to `runTask`.

Starting with this foundation (the monad of continuations, `mkTask` to build atomic tasks, and `runTask` to pull a behavior out of the task monad), we will now add some new features.

In the previous example, the robot description had to be passed explicitly into each part of the controller. We can pass this description implicitly rather than explicitly by building it into the task monad directly. That is, we want to pass the robot description to `runTask` and then have it appear wherever needed without adding extra `r` parameters to everything. In particular, the place we really need it to appear is `mkTask`, since the behavior and event are generally functions of the current robot.

We define a new type to encapsulate *task state*:

```
data TaskState =
  TaskState {taskRobot :: Robot}
```

For now, our state has only one element: the current robot. The type `TaskState` is defined using Haskell record syntax, which here defines `taskRobot` as a selector function to extract the robot from the task state. As more components are added to the task state, the definition of `TaskState` will change but code referring to state values will remain unaltered. We now add this state to the definition of `Task`. A task is given an initial state (the first parameter) and then passes a (potentially updated) state to the continuation carrying the next task:

```
data Task a b =
  Task (TaskState ->
    (TaskState -> b -> a) -> a)
instance Monad Task where
  (Task f) >>= g =
    Task (\ts c ->
      f ts (\ts' r ->
        unTask (g r) ts' c))
  return k =
    Task (\ts c -> c ts k)
```

Again, this instance definition is “off the shelf”: a standard combination of continuations and state. The `runTask` function now needs an initial state to pass into the first task. The definition of `runTask` is now:

```
runTask :: TaskState -> Task a b -> a -> a
runTask ts (Task t) finalB =
  t ts (\_ _ -> finalB)
```

In general, the call to `runTask` will need to fill in initial values for all components of the task state.

We've put `TaskState` into the monad, but how can we get it back out again? That is, how can tasks access information inside `TaskState`? These monadic operators directly manipulate the current `TaskState`:

```
getTaskState :: Task a TaskState
getTaskState = Task (\ts c -> c ts ts)
setTaskState :: TaskState -> Task a ()
setTaskState ts = Task (\_ c -> c ts ())
```

We also make the state available to tasks defined via `mkTask`. The argument to `mkTask` is now a function from the current task state onto the behavior and event defining the task:

```
mkTask :: (TaskState ->
  (Behavior a, Event b)) ->
  Task a b
mkTask f =
  Task (\ts c ->
    let (b,e) = f ts in
    b 'untilB' e ==> c)
```

The definitions in the previous example are now simplified: the robot is propagated to the tasks implicitly rather than explicitly:

```
goAhead, turnRight, runAround ::
  Task WheelControlB ()

goAhead =
  mkTask (\ts ->
    let r = taskRobot ts in
    (pairB 10 0,
     predicate (frontSonar r < 20)))

turnRight =
  mkTask (\ts ->
    let r = taskRobot ts in
    (pairB 0 0.5,
     predicate (frontSonar r > 30)))

runAround =
  do goAhead
  turnRight
  runAround -- loop forever
```

```
main =
  runController
    (\r -> runTask
      (TaskState {taskRobot = r})
      (runAround r)
      undefined)
```

Note that the composite task, `runAround`, is not aware of the propagation of the task state. We could have retained the old `mkTask` (without the `TaskState` argument) for compatibility but have chosen not to. This change could, though, have been made without invalidating any user code.

We have, so far, exploited well known monadic structures for continuations and state. One more basic monadic construction is of use: exceptions. With exceptions, tasks of type `Task a b` may succeed, returning a value of type `b`, or fail, raising an exception of type `RoboErr`. This is reflected in a new definition of the `Task` type in which a task may return either its terminating event value, `b`, or an error value, of type `RoboErr`.

```
data Task a b =
  Task (RState ->
    (RState -> (Either RoboErr b) -> a) -> a)
```

These primitives raise and catch exceptions:

```
taskCatch ::
  Task a b ->
  (RoboErr -> Task a b) ->
  Task a b
taskError :: -- Raise an error
  RoboErr -> Task a b
```

We omit the definitions of these primitives and the modified `Monad` instance; these are standard constructions along the lines of those in [18]. However, we will examine the changes needed to `mkTask`. That is, the `Monad` instance itself is essentially independent of the underlying domain, defined in terms of standard monad constructions and unspecific to robotics while the task creator, however, is very domain-specific and must be modified to account for the presence of exceptions.

Here is a new version of `mkTask` that adds an error event to the basic definition of a task. We use a slightly different name, `mkTaskE`, so that the old interface, `mkTask`, remains valid. The new definitions are:

```

mkTask ::
  (TaskState -> (Behavior a, Event b)) ->
  Task a b
mkTask f =
  mkTaskE (\ts -> let (b,e) = f ts in
                (b, e, neverE))

mkTaskE ::
  (Robot ->
   (Behavior a, Event b, Event RoboErr)) ->
  Task a b
mkTaskE f =
  Task (\ts c ->
    let (b,e,err) = f ts in
      b 'untilB' ((e ==> Right) .|.
                  (err ==> Left))
    ==> c)

```

The only change here is that the terminating event is either the normal exit event with the `Right` constructor added or an error event, as tagged by `Left`. The `.|.` operator is FRP construct that merges events, taking the first one to occur. The `==>` operator modifies the value of an event, so if `err` has type `Event RoboErr`, then `err ==> Right` has type `Event (Either a RoboErr)`. The constructors `Left` and `Right` define the `Either` type.

Next, consider a task such as “turn 90 degrees right”. Can we encode this easily as a `Frob` task? Not yet! The problem is that a task don’t know the orientation of the robot at the start of the task. We can build a control system to turn to a specified heading, but how do we know what the goal should be? The answer lies in the task state. During task transitions (the `untilB` in `mkTaskE`), we should also take note of where the robot is, which way its pointing, and other useful information.

In this simplified example, we capture the current robot location when moving from task to task. The monad remains unchanged (except for a new field in the `TaskState` structure) but the task builder must be modified as follows:

```

type RobotStatus = Point2

snapRobot ::
  Event a -> Robot -> Event (a,Radians)
snapRobot e r =
  e 'snapShot' (orientationB r)

mkTaskE f =
  Task (\ts c ->
    let (b,e,err) = f ts in
      b 'untilB'

```

```

((e ==> Right .|. err ==> Left)
 'snapRobot' (tsRobot ts))
==> (\(res,rstate) ->
  c (addRstate ts rstate) res)

```

The terminating event of the behavior is augmented with the state of the robot at the time of the event by the the `snapShot` function, a primitive defined FRP to capture the value of a behavior at the time of an event occurrence. Tasks will now find this initial orientation as part of the task state. This, a “turn right” task would be as follows:

```

turnRight =
  mkTask (\ts ->
    let goal =
      initialOrient ts + 90 in ...)

```

Empty tasks (those defined as a `return`) pass the state onto the next task unchanged.

3.2 Task Transformations

Having described the elementary task operations in detail, we now examine, briefly, other task operations. Given an existing task, what useful task transformations can be implemented? Examples include:

```

addError      ::
  Event RoboErr -> Task a b -> Task a b
timeLimit     ::
  Time -> Task a b -> Task a (Maybe b)
withB         ::
  (TaskState -> Behavior a) ->
  Task b c ->
  Task b (c,a)
withExit      ::
  Event a -> Task b c -> Task b a
withMyResult  ::
  (a -> Task a b) -> Task a b
withFilter    ::
  (a -> a) -> Task a b -> Task a b
withPicture   ::
  Behavior Picture -> Task a b -> Task a b

```

The operation of many of these functions is obvious from the type signature: an illustration of the value of polymorphic type signatures as documentation. While the full implementations of these functions is beyond the scope of this paper, we will provide a basic outline of each of these functions and how they are supported by the `Task` monad.

Most of the interesting semantic extensions to the system involve the basic definition of an atomic task. By bringing values from the task state into this definition, we can parameterize sequences of tasks rather than atomic ones. For example, consider `addError`: this function adds a new error event to an existing task. Note that the error event specified in `mkTaskE` applies only to an atomic task. The task passed to `addError`, however, may consist of many sequenced subtasks. The definition of `addError` looks something like this:

```
addError err ts
do oldErr <- previous global error event
  let newErr = err .|. oldErr
  place newErr into the task state
  execute ts
  restore prior global error event
```

Of course, `mkTask` must be changed too. The event `err` now becomes `err .|. getGlobalErr ts`: the error event in the task state must be included in the error condition in the `untilB`. This sort of "scoped reactivity" is not easily expressed in basic FRP; using the task monad makes it much easier to implement this feature.

The `timeLimit` function aborts a task if it does not complete within a specified time. It is implemented using `addError` to attach an event that to the associated task which occurs at the specified time. This requires both the exception and state capabilities of the underlying monad.

The `withB` function defines a behavior to run in parallel with a task. When the task exits, the value of the behavior is added to the task's result value. This is implemented by building a task that attaches a snapshotting function to the incoming continuation.

The `withExit` function aborts a task upon an event. If the task completes before the aborting event, an error occurs. This is implemented directly in FRP by `untilB`, as in this simplified definition:

```
withExit e (Task t) =
  Task (\ts c -> t ts err 'untilB' e ==> c)
  where err = error "Premature task exit"
```

This function is implemented directly at the continuation level.

The `withMyResult` function it allows a task to observe its own result, which is often needed in the differential equations that define a controller.

```
withMyResult f =
  Task (\ts c -> let r = (unTask t) ts c
                  t = f r in
          e)
```

Another place in which the atomic definition of a task may be further parameterized is the resulting behavior. That is, instead of

```
b 'untilB' (e ...)
```

we modify the resulting behavior using a filter in the task state:

```
((getfilter ts) b) 'untilB' (e ...)
```

This is implemented in a manner similar to `withError`, using

```
withFilter ::
  (a -> a) -> Task a b -> Task a b
```

Finally, we discuss a more global change to the task structure. Debugging controllers is difficult: it is hard to visualize the operation of a control system based on printing out numbers on the screen as the controller executes. A much better debugging technique is to display diagnostic information graphically in the robot simulator, painting various cues onto the simulated world to graphically convey information. For this, we augment the behavior defined by a task to include an animation. That is, using the task monad we provide an implicit channel to convey diagnostic information along with the behavior. This modification requires changes to the definition of `Task`: `a` is replaced by `(a, Behavior Picture)`, the type now produced by `runTask`. This change does not affect user-level code; the extra picture is implicit in every task. When a program is running on a real robot, the animation coming out of `runTask` is ignored.

This is the definition of `withPicture`:

```
withPicture ::
  Behavior Picture -> Task a b -> Task a b
withPicture p t =
  addFilter (addPicture p) t
```

The `addPicture` function introduces an additional picture to an augmented behavior. For example,

this function makes `driveToGoal` easier to understand in simulation:

```
driveToWithPicture goal =
  driveTo goal 'withPicture'
  paintAt goal (withColor red circle)
```

3.3 Parallel Tasks

So far, we have only modified tasks or combined them sequentially. Now, we wish to combine tasks by merging the results of multiple tasks running in parallel. The `withTask` function is the basic primitive for combining tasks in parallel:

```
withTask :: (t2b -> Task t1b t1e) ->
  (t1b ->
    Event (Either RoboErr t2e) ->
    Task t2b t2e) ->
  Task t2bt2e
```

This initiates two tasks, each observing the behavior defined by the other. The termination of the first task, either through an exception or normal termination, may be observed by the other task as an event.

The code associated with `withTask` is as follows:

```
withTask t1f t2f =
  Task (\ts c ->
    let t1e = makeNewEvent
        t1b = (unTask t1)
              (cloneState ts)
              (sendTo t1e)
        t2b = (unTask t2) ts c
        t1  = t1f t2b
        t2  = t2f t1b t1e in
    t2b)
```

Note the somewhat imperative treatment of the terminating event of `t1`. The `sendTo` and `makeNewEvent` functions exploit FRP internals, an expedient but semantically unusual way of dealing with events. The `sendTo` function becomes an undefined behavior after sending the termination message; reference to the value of `t1` after this event will result in a runtime error.

More importantly, notice the treatment of the task state. The task `t1` needs to receive a “fresh copy”

of the overall task state. Local error handlers and filters are removed from the state so that only `t2` inherits these.

4 Examples

Assessing a DSL is often difficult; different DSL's are designed with different goals. Since our goal is to build a language that is declarative and descriptive, we choose to assess it with examples of code rather than performance figures. These examples are chosen to demonstrate expressiveness rather than computational speed.

4.1 The BUG Algorithm

First, we demonstrate the use of tasks to implement a well known control strategy. BUG [7] is an algorithm to navigate around obstacles to a specified goal. When an obstacle is encountered, the robot circles the obstacle, looking for the point closest to the goal and then returns to this point to resume travel. The following code skeleton implements BUG in terms of two primitive behaviors: driving straight to a goal, and following a wall. The `driveTo` task returns a boolean: `true` when the goal is reached, `false` when the robot is blocked. The `followWall` runs indefinitely, traveling in circles around an obstacle. If for some reason the wall disappears from the sonars, this task raises an exception.

```
-- Basic tasks and events (not shown)
followWall ::
  Task WheelControlB ()
driveTo    ::
  Point2 -> Task WheelControlB Bool
atPlace    ::
  Robot -> Point2 -> Event ()

bug        ::
  Point2 -> Task WheelControlB ()
bug g      =
  taskCatch (bug g) -- restart on error
  (do finished <- driveTo g
    if finished
      then return ()
      else goAround g)

goAround   ::
  Task WheelControlB ()
```

```

goAround g =
  do closestPoint <- circleOnceP g -- circle
    circleTo closestPoint -- then back
    bug g -- restart

circleOnceP ::
  Point2 -> Task WheelControlB Point2
circleOnceP g =
  do (_,p) <- withB closestP (circleOnce g)
    return p
  where closestP ts =
    let r = taskRobot ts in
    (distance (place r) g) 'atMin' (place r)

circleOnce =
  do ts <- getTaskStatus
    let initp = initialPlace ts
    r = taskRobot ts
    -- get away from initial place
    timeLimit followWall 5
    followWall 'withExit' (atPlace initp)

```

This definition is quite close to the informal definition of BUG. Some necessary details have been filled in: what to do if the wall disappears from the sensors (this is caught at the top level and restarts the system), how to circle (travel for 5 seconds to get away from the start point and then continue until the start point is re-attained).

4.2 A Process Architecture

As another example, consider Lyons' approach of capturing robotic action plans as networks of concurrent processes [8, 9]. Frob tasks can easily mimic Lyons' processes. His conditional composition operation is identical to $\gg=$ in the task monad with exceptions. Of more interest is parallel composition: his $P \mid Q$ executes processes P and Q in parallel, with ports connecting P and Q . This can be directly implemented with `withTask`. His disabling composition, $P \# Q$ is also contained in `withTask`, however `withError` is also needed to correctly disable the resulting task when one task aborts.

The lazy evaluation semantics of Frob permits the following operations (synchronous concurrent composition and asynchronous concurrent composition) to be implemented directly:

```

P <> Q = do { v<-P; Q; (P<>Q) }
P >< Q = do { v<-P; (Q | (P><Q)) }

```

5 Conclusions

We have demonstrated both a successful DSL for robotic control and shown how a set of tools developed in the functional programming community enable the construction of complex DSLs with relatively little effort.

Assessing a DSL (or any programming language) is difficult at best. We feel the success of Frob is demonstrated in a number of ways:

- Users from outside of the FP community find that Frob is easy to use and well-suited to the task of robot control. The abstractions supplied by Frob may be understood at an intuitive level. While there is a definite learning curve, especially with respect to the Haskell type system, users soon become accustomed to polymorphic typing and find Haskell types much more descriptive than those of object oriented systems.
- We have encoded a number of well-known algorithms and architectural styles in Frob and found the results to be elegant, concise, and modular.
- As an embedded DSL, Frob inter-operates easily with other DSLs. Preliminary work on combining Frob with FVision (a very different DSL for vision processing) suggests that at least some DSLs may be combined to great advantage.
- Monads support relatively painless evolution of DSL semantics. DSLs are, rather naturally, somewhat of a moving target: as domain engineers and DSL implementors work together, improvements in semantic expressiveness are continually being developed. The monadic framework has allowed this semantic evolution to proceed without requiring constant rewriting of existing code.

Some issues are still unresolved or unaddressed: we have not yet ported the system to new types of robots or implemented systems in which system performance is critical. We also have yet to experiment with multi-robot systems. Frob has not been used in any way that taxed system performance; the data rates from our sensors are so low that the controller has relatively little work to do.

One particularly large part of this domain that has yet to be addressed is real time. For example, we cannot express the priorities of various activities, allowing Frob to direct resources toward more critical systems when needed. No guarantees are made with respect to responsiveness or throughput. We expect that Frob is capable of addressing these issues but much more complex analysis and code generation may be required. However, for high level system control (as exemplified by the BUG algorithm) these issues are of less importance.

We have used Frob to teach an undergraduate robotics course. Frob was quite successful in allowing assignments that traditionally required many pages of C++ code to be programmed in only a page or two. While there was an admittedly steep learning curve, students eventually became quite productive. The addition of graphic feedback in the simulator was especially useful to them.

Turning to the issue of DSL construction, this research shows that monads are an important tool for attaining program modularity. The definition of task monad evolved significantly over the term but the interfaces remained the same, allowing all student code to run unchanged as Frob evolved. Monads effectively hide potentially complex machinery and provide a framework whereby new functionality can be added to a system with minimal impact on existing code.

All Frob software, papers, and manuals are available at <http://haskell.org/frob>. Nomadics has agreed to license their simulator to Frob users at no cost, allowing our software to be used by those without real robots to control.

6 Acknowledgements

This research was supported by NSF Experimental Software Systems grant CCR-9706747.

References

[1] Rodney Brooks. A robust layered control system for a mobile robot. *IEEE Trans. on Robotics and Automation*, 2(1):24–30, March 1986.

[2] E. Coste-Manière and B. Espiau, editors. *International Journal of Robotic Research, Special Issue on Integrated Architectures for Robot Control and Programming*, volume 17:4, 1998.

[3] Conal Elliott. Composing reactive animations. *Dr. Dobbs' Journal*, July 1998. Extended version with animations at <http://research.microsoft.com/conal/fran/{tutorial.htm,tutorialArticle.zip}>.

[4] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 163–173, June 1997.

[5] V. Hayward and J. Lloyd. *RCCL User's Guide*. McGill University, Montréal, Québec, Canada, 1984.

[6] K. Konolige. Colbert: A language for reactive control in sapphira. In G. Brewka, C. Habel, and B. Nebel, editors, *Advances in Artificial Intelligence*, volume 1303 of *Lecture Notes in Computer Science*. Springer, 1997.

[7] V.J. Lumelsky and A.A. Stepanov. Dynamic path planning for a mobile automaton with limited information on the environment. *IEEE Trans. on Automatic Control*, 31(11):1058–63, 1986.

[8] D. Lyons and M. Arbib. A formal model of computation for sensor-based robotics. *IEEE Trans. on Robotics and Automation*, 6(3):280–293, 1989.

[9] Damian M. Lyons. Representing and analyzing action plans as networks of concurrent processes. *IEEE Transactions on Robotics and Automation*, 9(7):241–256, June 1993.

[10] J.L. Mundy. The image understanding environment program. *IEEE EXPERT*, 10(6):64–73, December 1995.

[11] Pattis, R et al. *Karel the Robot*. John Wiley & Sons, 1995.

[12] J. Peterson, G. Hager, and P. Hudak. A language for declarative robotic programming. In *Proceedings of IEEE Conf. on Robotics and Automation*, May 1999.

[13] J. Peterson, P. Hudak, and C. Elliott. Lambda in motion: Controlling robots with haskell. In *Proceedings of PADL 99: Practical Aspects of Declarative Languages*, pages 91–105, Jan 1999.

- [14] John Peterson and Kevin Hammond. Haskell 1.4: A non-strict, purely functional language. Technical Report YALEU/DCS/RR-1106, Department of Computer Science, Yale University, May 1997.
- [15] B. Shimono. *VAL: A Versatile Robot Programming and Control Language*. IEEE Press, 1986.
- [16] D.B. Stewart and P.K. Khosla. The chimera methodology: Designing dynamically reconfigurable and reusable real-time software using port-based objects. *International Journal of Software Engineering and Knowledge Engineering*, 6(2):249–277, 1996.
- [17] R. H. Taylor, P.D. Summers, and J. M. Meyer. AML: A manufacturing language. *Int. J. of Robot Res.*, 1(3):3–18, 1982.
- [18] P. Wadler. The essence of functional programming. In *Proceedings of ACM Symposium on Principles of Programming Languages*. ACM SIGPLAN, January 1992.

Domain Specific Embedded Compilers

Daan Leijen and Erik Meijer

University of Utrecht

Department of Computer Science

POBox 80.089, 3508 TB Utrecht, The Netherlands

{daan, erik}@cs.uu.nl, <http://www.cs.uu.nl/~{daan, erik}>

Abstract

Domain-specific embedded languages (DSELs) expressed in higher-order, typed (HOT) languages provide a composable framework for domain-specific abstractions. Such a framework is of greater utility than a collection of stand-alone domain-specific languages. Usually, embedded domain specific languages are build on top of a set of domain specific primitive functions that are ultimately implemented using some form of foreign function call. We sketch a general design pattern for embedding client-server style services into Haskell using a domain specific embedded compiler for the server's source language. In particular we apply this idea to implement Haskell/DB, a domain specific embedded compiler that dynamically generates of SQL queries from monad comprehensions, which are then executed on an arbitrary ODBC database server.

1 Introduction

Databases are ubiquitous in computer science. For instance, a web site is usually a fancy facade in front of a conventional database, which makes the information available in a convenient browsable form. Sometimes, servers are even running directly on a database engine that generates pages from database records on-the-fly. Hence it is not surprising that database vendors provide hooks that enable client applications to access and manipulate their database servers. On Unix platforms this is usually done via ODBC, under Windows there are confusingly many possibilities such as ADO, OLE DB, and ODBC.

What is common to all the above database bindings is that queries are communicated to the database

as unstructured strings (usually) representing SQL expressions. This low-level approach has many disadvantages:

- Programmers get no (static) safeguards against creating syntactically incorrect or ill-typed queries, which can lead to hard to find runtime errors.
- Programmers have to distinguish between at least two different languages, SQL and the scripting language that generates the queries and submits them to the database engine (Perl, Visual Basic). This makes programming needlessly complex.
- Programmers are exposed to the accidental complexity and idiosyncrasies of the particular database binding, making code harder to write and less robust against the vendor's fads [1].

We argue that domain-specific embedded languages [9] (DSELs) expressed in higher-order, typed (HOT) languages, Haskell [10] in our case, provide a composable framework for domain-specific abstractions that is of greater utility than a collection of stand-alone domain-specific languages:

- Programmers have to learn only one language, domain specific abstractions are exposed to the host language as extension libraries.
- In many cases it is possible to present libraries using a convenient domain specific syntax.
- It is nearly always¹ possible to guarantee that programmers can only produce syntactically correct target programs, and in many cases we are able to impose domain specific typing rules.

¹For instance \perp is a value of every type in Haskell, so cannot prevent programmers from producing infinite or partially defined values.

- Programmers can seamlessly integrate with other domain specific libraries (e.g. CGI, mail), which are accessible in the same way as any other library. This is a largely underestimated benefit of using the embedded approach. Connecting different domain specific languages together is usually quite difficult.
- Programmers can leverage on existing language infrastructure such as the module and type system and the built-in abstraction mechanisms.

Note that the ideas underlying our thesis date way back to 1966 when Peter Landin [12] already observed that all programming languages comprise a domain independent linguistic framework and a domain dependent set of components. What is new in this paper is that we show how to embed the terms and the type system of another (domain specific) *programming language* into the Haskell framework, which dynamically *compiles* and *executes* programs written in the embedded language. Moreover, no changes to the syntax or additions of primitives were needed to embed the language in Haskell.

1.1 Overview

We begin by giving a minuscule introduction to Haskell and a crash course in relational databases, and we show how a typical Visual Basic and a typical Haskell program would access a relational database. Next we sketch a general design pattern for term- and type-safe embedding client-server style services into Haskell using an evaluator for a subset of SQL expressions as an example server. We then turn our attention to the more challenging task of embedding a database server in Haskell. Section 7 contrasts the Haskell and Visual Basic implementations of a example web-page that generates HTML from a database of exam marks. We finish with some conclusions and some ideas for future work.

2 Minuscule introduction to Haskell

The main virtue of a functional language is that functions are first-class citizens that can be stored in lists, or passed as arguments to and returned from other functions. To emphasize the fact that functions of type $a \rightarrow b$ have the same status as

any other kind of value, we usually write them as lambda-expressions $f = \lambda a \rightarrow b$ instead of the more common Haskellish notation $f\ a = b$.

Function application is given by juxtaposition in Haskell and associates to the left. Thus when we have the three argument function `line`:

```
line = \a -> \b -> \x -> a*x + b
```

the application `line 2 3` denotes the single argument function $\lambda x \rightarrow 2*x + 3$. The type of the `line` function can be specified explicitly:

```
line :: Int -> Int -> Int -> Int
```

The type shows that `line` takes three `Int` arguments and returns an `Int`.

Case expressions are used to define functions by case distinction, for instance the factorial function can be defined in Haskell as

```
fac :: Num a => a -> a
fac = \n ->
  case n of
    { 0 -> 1
    ; n -> n * fac (n-1)
    }
```

In Haskell, polymorphic types can be constrained by means of type contexts. The given type for the factorial function `fac :: Num a => a -> a -> a` says that function `fac` has type $a \rightarrow a \rightarrow a$ for all types a that are instances of the `Num` class. Unsurprisingly it is the case that `Num Int`, `Num Float` and `Num Double` are all true.

We will represent database rows by extensible records, an experimental feature that is currently only supported by the the *TREX* extension of the Hugs implementation of Haskell [8]. A record is nothing more than an association list of field-value pairs. For instance the record $(x = 3, \text{even} = \text{False}) :: \text{Rec } (x :: \text{Int}, \text{even} :: \text{Bool})$ has two fields, `x` of type `Int` and `even` of type `Bool`. A record of type `Rec r` can be *extended* by a field `z` provided that `z` does not already occur in `r`. The fact that record `r` should lack field `z` is indicated by the constraint `r\z`, thus the type of a function that adds an `foo` field to a record becomes:

```
extendWithFoo :: r\foo =>
```

```

a -> Rec r -> Rec (foo :: a | r)
extendWithFoo = \a -> \r -> (foo = a | r)

```

Unfortunately, labels are not (yet) first class values in TREX, so we *cannot* write a generic function that extends a given record with a new field:

```

-- WRONG
extendWith = \(f,a) -> \r -> (f = a | r)

```

In our case the lack of first class labels means that we have to repeat a lot of code that only differs in the names of some labels. Another deficiency of the current implementation of TREX records is the fact that it is impossible to formulate a constraint on all the values in a given record, for instance, we would like to constrain a record to contain only values on which equality is defined. Currently, there is just one built-in constraint `ShowRecRow r` that indicates that all values in row `r` are in the `Show` class.

When interacting with the outside world or accessing object models, we have to deal with side-effects. In Haskell, effectful computations live in the `IO` monad [15]. A value of type `IO a` is a *latently* effectful computation that, *when executed*, will produce a value of type `a`. For instance, the command `getChar :: IO Char` will read a character from the standard input when it is executed.

Like functions, latently effectful computations are first class citizens that can be passed as arguments, stored in list, and returned as results. For example `putChar :: Char -> IO ()` is a function that takes a character and then returns a computation that, when executed, will print that character on the standard output.

Effectful computations can be composed using the `do{}`-notation. The command `do{ a <- ma; f a } :: IO b` is a latent computation, that, when executed, first executes the command `ma :: IO a` to obtain a value `a :: a`, passes that to the action-producing function `f :: a -> IO b` and then executes `(f a)` to obtain a value of type `IO b`. For example, when executed, the command:

```

do{ c <- getChar
  ; putChar c
}

```

reads a character from the standard input and copies it to the standard output.

The usefulness of monads goes far beyond input/output, many other type constructors are monads as well. In section 6 we will define the `Query` monad that allows us to write queries using the same `do{}` notation that we introduced here for `IO`-computations.

In this paper we adopt style conventions that emphasize when we are dealing with effectful computations. Specifically, all expressions of monadic type (such as `IO` and `Query`) are written with an explicit `do{}`. To reflect the influence of the OO style, we will use postfix function application `object#method = method object` to mimic the `object.method` notation. Together with the convention for writing functions as lambda-expressions, this results in highly stylized programs from which it is easy to tell the type of an expression by looking at its syntactic shape.

3 A crash course in relational databases

In a relational database [5], data is represented as sets of tuples. For example take the following database *Rogerson* of objects and some of their properties [17]:

Object	Edible	Inheritance	President
Rich people	False	False	True
Bean plants	True	False	False
CORBA	False	True	False
COM	False	False	False

We can conclude from this table that bean plants are edible, and that rich people can run for president. We can query the database more systematically using relational algebra.

The *selection* operator σ specifies a subset of rows whose attributes satisfy some property. For example we can eliminate all entries for objects that can run for president from the database using the expression $\sigma_{President=False} \text{Rogerson}$:

Object	Edible	Inheritance	President
Bean plants	True	False	False
CORBA	False	True	False
COM	False	False	False

The *projection* operator π specifies a subset of the columns of the database. For example, we can return all objects that are edible using the query $\pi_{Object} (\sigma_{Edible=True} Rogerson)$

Object
Bean plants

Another typical operation on relations is join \bowtie that combines two relations by merging tuples whose common attributes have identical values. Hence, if we join the *Presidents* table

Name	President
Starr	False
Clinton	True

with the *Rogerson* table using $\pi_{Name, Object} (Presidents \bowtie Rogerson)$ we get a table with the name and object description of people that can run for president:

Name	Object
Clinton	Rich people

3.1 The SQL way: SELECT statement

SQL is the defacto standard programming language to formulate queries over relational databases. The SQL query

```
SELECT columns
FROM tables
WHERE criteria
```

combines selections, projections and joins in one powerful primitive. The SELECT clause specifies the columns to project, the FROM clause specifies the tables where the columns are located and the WHERE clause specifies which rows in the tables should be selected.

The query $\sigma_{President=False} Rogerson$ is expressed in SQL as:

```
SELECT *
FROM Rogerson AS r
WHERE r.President = FALSE
```

The query $\pi_{Object} (\sigma_{Edible=True} Rogerson)$ becomes:

```
SELECT r.Object
FROM Rogerson AS r
WHERE r.Edible = TRUE
```

The query $\pi_{Name, Object} (Presidents \bowtie Rogerson)$ is expressed as:

```
SELECT p.name, r.Object
FROM Rogerson AS r, Presidents AS p
WHERE r.President = p.President
```

3.2 The VB way: Unstructured strings

The common way to do query processing from Visual Basic is to build an unstructured string representing the SQL query and submitting that to a database server object (we will discuss the ADO object model in more detail in section 6.1). So for instance, The query $\sigma_{President=False} Rogerson$ is expressed in Visual Basic as:

```
Q = "SELECT *"
Q = Q & "FROM Rogerson AS r"
Q = Q & "WHERE r.President = FALSE"

Set RS = CreateObject("ADODB.Recordset")
RS.Open Q "Rogerson"

Do While Not RS.EOF
    Print RS("Object")
    Print RS("Edible")
    Print RS("Inheritance")
    Print RS("President")
    RS.MoveNext
Loop
```

3.3 The FP way: Comprehensions

Within the functional programming community, people have argued that (list) comprehensions are a good query notation for database programming languages [2]. For example using the comprehension notation supported by Haskell/DB, the query $\sigma_{President=False} Rogerson$ can be expressed as:

```
do{ r <- table rogerson
    ; restrict
```

```

    (r!president .==. constant False)
; return r
}

```

The query $\pi_{Object}(\sigma_{Edible=True} Rogerson)$ becomes

```

do{ r <- table rogerson
; restrict (r!edible .==. constant True)
; project (object = r!object)
}

```

Queries that use projections and joins such as the following $\pi_{Name,Object}(Presidents \bowtie Rogerson)$ are harder to formulate because we have to indicate explicitly on which common fields to compare and how to create the resulting tuple:

```

do{ r <- table rogerson
; p <- table president
; restrict (r!president .==. p!president)
; project (name = p!name, object = r!object)
}

```

The comprehensions are fully *typed* and automatically translated into correct SQL strings which are sent to a low-level database server. This paper describes not only how we did this for SQL but also tries to give a general recipe for embedding languages into a strongly typed language.

Let's put the question of embedding SQL aside until section 6 and first look how we in general can embed languages into Haskell.

4 Term embedding

Although SQL is embedded in this specific case, there is a general strategy for embedding services in a HOT language. We will illustrate this using a simple SQL expression service as an example. In SQL, expressions are used (amongst others) in the search conditions of WHERE clauses to perform computations and comparisons on columns and values.

4.1 SQL expression server

Lets assume that the SQL expression server provides us with the following interface for evaluating expressions (described in IDL):

```

interface IServer
{ void SetRequest([in,string] char* expr);
; void GetResponse([out] char* result);
}

```

Although simple, the IServer interface captures the essence of many dynamic services such as a desk calculator, finger, HTTP, ftp, NNTP, DNS, ODBC, ADO and similar information servers.

From Haskell, we can access the IServer interface using the functions `setRequest`, and `getResponse` that are automatically generated by our *H/Direct* IDL compiler [6]:

```

setRequest :: String -> IServer s -> IO ()
getResponse :: IServer s -> IO String

```

We are now able to write an evaluator function that takes an expression, sends it to the server and returns the result:

```

runExpr :: String -> IO Int
runExpr = \expr ->
do{ server <- createObject "Expr.Server"
; server # setRequest expr
; result <- server # getResponse
; return (read result)
}

```

This is essentially the kind of interface that is in use now with SQL server protocols as ODBC or ADO. An unstructured SQL string is directly sent to the server. The problem is that there is nothing that prevents the programmer from sending invalid strings to the server, leading to errors at run-time and/or unpredictable behavior of the server. Clearly, this is unacceptable in critical business applications.

4.2 Abstract syntax

To prevent the construction of syntactically incorrect expressions, we define an *abstract syntax* for the terms of the input language of the specific server we are targeting, together with a "code generator" to map abstract syntax trees into the concrete syntax of the input language.

The abstract syntax `PrimExpr` simply defines literal constants, and unary and binary operators. (In section 6 we will add row selection):

```
data PrimExpr
  = BinExpr  BinOp PrimExpr PrimExpr
  | UnExpr   UnOp  PrimExpr
  | ConstExpr String
```

```
data BinOp
  = OpEq | OpAnd | OpPlus | ...
```

Types `UnOp` and `BinOp` are just enumerations of the unary and binary operators of SQL expressions.

Writing expressions directly in abstract syntax is not very convenient, so we provide combinators to make the programmer's life more comfortable. Each SQL operator is represented in Haskell by the same operator surrounded with dots. Some definitions are²:

```
constant :: Show a => a -> PrimExpr
(+. .)   :: PrimExpr -> PrimExpr -> PrimExpr
(==. .)  :: PrimExpr -> PrimExpr -> PrimExpr
(.AND. .) :: PrimExpr -> PrimExpr -> PrimExpr
```

Now we can write `constant 3 ==. constant 5` instead of the cumbersome `BinExpr OpEq (ConstExpr (show 3)) (ConstExpr (show 5))`. This is what *embedded* domain specific languages are all about!

4.3 Concrete syntax

In order to evaluate expressions, we must map them into the exact *concrete* syntax that is required by the server component.

The code generator for our expression server is straightforward; we print expressions into their fully parenthesized concrete representation by a simple inductive function:

```
pPrimExpr :: PrimExpr -> String
pPrimExpr = \e ->
  case e of
    { ConstExpr s
      -> s
    ; UnExpr op x
      -> pUnOp op ++ parens x
    ; BinExpr op x y
```

²The `constant` function is unsafe since any value in the `Show` class can be used. In the real library we introduce a separate class `ShowConstant` which is only defined on basic types.

```
-> parens x ++ pBinOp op ++ parens y
}
```

```
parens = \x -> "(" ++ pPrimExpr x ++ ")"
```

Normally however, this step will be more involved as we will see in the SQL example.

4.4 Embedding expressions

Now that we know how to construct expressions and how to generate code from them, we can rewrite the evaluator function to use the structured expressions:

```
runExpr :: PrimExpr -> IO Int
runExpr = \expr ->
  do{ server <- CreateObject "Expr.Server"
    ; server # setRequest (pPrimExpr expr)
    ; result <- server # getResponse
    ; return (read result)
  }
```

We can now use SQL expressions in Haskell as if they were built-in. Function `runExpr` will dynamically compile a `PrimExpr` *program* into *target code*, execute that on the expression server and coerce the result back into a Haskell integer *value*:

```
sum :: Int -> PrimExpr
sum = \n ->
  if (n <= 0)
  then (constant 0)
  else (constant n +. sum (n-1))

test = do{ runExpr (sum 10) }
```

5 Type embedding

The above embedding is already superior to constructing unstructured string to pass to the server because it is impossible to construct syntactically incorrect requests. However, it is still possible to construct *ill typed* request, as the following example shows:

```
do{ let wrong = (constant 3) .AND. (constant 5)
    ; result <- runExpr wrong
    ; print result
  }
```

Since the `PrimExpr` data type is completely untyped, we have no way to prevent the construction of terms such as `wrong` that might crash the server because the operands of the `AND` expression are not of boolean type.

5.1 Phantom types

We used abstract syntax trees to ensure that we can only generate syntactically correct request, but the billion dollar question of course is whether there is a similar trick to ensure that we can only generate type correct requests. Fortunately, the answer is yes! It is possible indeed to add an extra layer on top of `PrimExpr` that effectively serves as a type system for the input language of the expression server.

The trick is to introduce a new *polymorphic* type `Expr a` such that `expr :: Expr a` means that `expr` is an expression of type `a`. The type variable `a` in the definition of the `Expr` data type is only used to hold a type; it does not occur in the right hand side of its definition and is therefore never physically present:

```
data Expr a = Expr PrimExpr
```

Now we refine the types of the functions to construct values of type `Expr a` to encode the typing rules for expressions:

```
constant :: Show a => a -> Expr a
(.,.) :: Expr Int -> Expr Int -> Expr Int
(==.) :: Eq a => Expr a -> Expr a -> Expr Bool
(.AND.) :: Expr Bool -> Expr Bool -> Expr Bool
```

For example, the definition of `(==.)` is now:

```
(==.) :: Eq a => Expr a -> Expr a -> Expr Bool
(Expr x) ==. (Expr y)
    = Expr (BinExpr OpEq x y)
```

By making the `Expr` type an abstract data type, we ensure that only the primitive functions can use the unsafe `PrimExpr` type. If we now use these combinators to write `(constant 3) .AND. (constant 5)`, the *Haskell type-checker* will complain that the type `Expr Int` of the operand `(constant 2)` does not match the required type `Expr Bool`.

The typing of expressions via phantom type variables extends immediately to values built using Haskell primitives. Our example function `sum` for instance, now has type `sum :: Int -> Expr Int`.

Phantom type variables have many other exciting uses, for instance in encoding inheritance and typing pointers [7]. Later we will show how we use multiple phantom type variables to give a type safe encoding of attribute selection in records.

6 Embedding SQL

Armed with the knowledge of how to safely embed a simple language into Haskell, we return to our original task of embedding SQL into Haskell.

6.1 The SQL server

We will use ActiveX Data Objects (ADO) as our SQL server component. ADO is a COM [11] framework that can use any ODBC compliant database; MS SQL Server, Oracle, DB/2, MS Access and many others. The ADO object model is very rich but we will use only a tiny fraction of its functionality.

ADO represents a relation as a `RecordSet` object. It creates a set of records from a query via its `Open` method:

```
dispinterface Recordset {
    void Open
        ([in,optional] VARIANT Source,
         [in,optional] VARIANT ActiveConnection,
         [in,optional] CursorTypeEnum CursorType,
         [in,optional] LockTypeEnum LockType,
         [in,optional] long Options
        );
    Bool EOF();
    void MoveNext();
    Fields* GetFields();
}
```

The first argument of the `Open` method is the source of the recordset, which can be an SQL string or the name of a table or a stored procedure. The second argument of the `Open` method can be a connection string, in which case a new connection is made to

create the recordset, or it can be a Connection object that we have created earlier. In this paper, we will not use the other (optional) arguments of the `Open` method, hence we provide the following signature for `open`:

```
open :: (VARIANT src, VARIANT actConn) =>
      src -> actConn -> IRecordSet r -> IO ()
```

The `MoveNext`, `EOF` and `GetFields` methods allow us to navigate through the recordset. Their Haskell signatures are:

```
moveNext :: IRecordSet r -> IO ()
eof       :: IRecordSet r -> IO Bool
getFields :: IRecordSet r -> IO (IFields ())
```

The `Fields` interface gives access to all the fields of a row, they can be accessed either by position or by name:

```
disinterface Fields {
    long    GetCount();
    Field* GetItem([in] VARIANT Index);
};
```

Each `Field` object corresponds to a column in the `Recordset`:

```
disinterface Field {
    VARIANT GetValue();
    BSTR    GetName();
};
```

The `GetValue` property of a `Field` object can be used to obtain the value of a column in the current row. The `GetName` property returns the name of the field:

```
getValue :: VARIANT a => IField f -> IO a
getName  :: IField f -> IO String
```

Although the ADO object model is somewhat more refined than the expression evaluator example we have seen earlier, it does still fit the basic client-server framework. Requests are submitted via the `Open` method, and responses are inspected by iterating over the individual `Field` objects of the `Fields` collection.

6.2 Using the RecordSet in Haskell

In Haskell, we would like to abstract from iterating through the record set and access the result of performing a query as a list of fields. This faces us with the choice of either returning this list eagerly, or creating it lazily. In the former case, all fields are read into a list at once. In the latter case, the fields are encapsulated in a lazy stream where a field is read by demand.

Both functions are defined in terms of the function `readFields` that takes an IO-action transformer function as an additional argument:

```
readFields :: (IO a -> IO a)
            -> IRecordSet r -> IO [IFields ()]
readFields = \perform -> \records -> perform $
do{ atEOF <- records # eof
   ; if atEOF
   then do{ return [] }
   else do{ field <- records# etFields
            ; records#moveNext
            ; rest <- rs#readFields perform
            ; return ([field] ++ rest)
   }
}
```

By taking `perform` to be the identity, we get a function that reads the list of fields strictly, by taking `perform` to be the IO-delaying function `unsafeInterleaveIO` we obtain a function that reads the list of fields lazily.

A simple query evaluator can now be written as:

```
runQuery :: String -> IO [IFields ()]
runQuery = \query ->
do{ records <- createObject "ADO.RecordSet"
   ; records # open query Nothing
   ; fields <- records # readFields id
   ; return fields
}
```

Of course, this approach suffers from all the weaknesses described in section 4.

6.3 Abstract syntax

Just as in the previous example we will define and *abstract syntax* for expressing database operations.

Our language for expressing those operations will be the relational algebra. The code generator will take these expressions and translate them to the concrete syntax of SQL statements which preserve the semantics of the original expression.

The abstract syntax for the relation algebra becomes³:

```

type TableName = String
type Attribute = String
type Scheme    = [Attribute]
type Assoc     = [(Attribute, PrimExpr)]

data PrimQuery
  = BaseTable TableName Scheme
  | Project   Assoc   PrimQuery
  | Restrict  PrimExpr PrimQuery
  | Binary    RelOp   PrimQuery PrimQuery
  | Empty

data RelOp
  = Times | Union | Intersect
  | Divide | Difference

data PrimExpr
  = AttrExpr Attribute
  | ConstExpr String
  | BinExpr  BinOp  PrimExpr PrimExpr
  | UnExpr  UnOp   PrimExpr

```

For example, the relational expression that returns all objects that are edible: $\pi_{Object}(\sigma_{Edible=True} \text{Rogerson})$ can be expressed in our abstract syntax as:

```

Project [("Object", AttrExpr "Object")]
  (Restrict (BinExpr OpEq (AttrExpr "Edible")
                        (ConstExpr "True")))
    (BaseTable "Rogerson"
      [ "Object", "Edible"
      , "Inheritance", "President"
      ]
    )
)

```

6.4 Concrete syntax

It is straightforward to generate concrete SQL statements from the `PrimQuery` data type, although special care has to be taken to preserve the correct semantics of the relational algebra due to the idiosyncrasies of SQL. The use of the relational algebra as

³The `Project` constructor actually does both projection and renaming.

an intermediate language allows us to target a wide range of different database languages. We are planning to add bindings to other dialects of SQL and languages as ASN.1.

Besides being portable, the simple semantics of the relational algebra allows us to perform a powerful set of optimizations quite easily before transforming the expression to concrete syntax. Many times the SQL server is not capable of doing these transformations due to the complex semantics of SQL. Another benefit is that we can add operations like table comparisons which are very hard to express in languages like SQL, but easy to generate from a relational expression.

6.5 Towards comprehensions

We could proceed as in our earlier example and define some friendly combinators for specifying relational expressions as we did in our previous example. However there is a serious drawback to using relational expressions directly as our programming language. In the relational algebra, attributes are only specified by their name. There is no separate binding mechanism to distinguish attributes from different tables. Suppose we take the cartesian product of a relation with itself. In SQL we could write:

```

SELECT X.Name, X.Mark
FROM Students As X, Students As Y
WHERE X.Mark = Y.Mark
AND X.Name <> Y.Name

```

But in the relational algebra, we are unable to do this since there are common attributes like `Name` and `City` which lead to ambiguity. To take the cartesian product, one relation needs to rename those attributes. The join (\bowtie) operator is especially introduced to make it easier to specify the most common products where renaming would be necessary. Besides only covering the most common cases, it is notoriously hard to typecheck join expressions [3] and we haven't found a way to embed those typing rules within Haskell.

However, why not use the same approach as SQL? We will introduce a binding mechanism (monad comprehensions) for qualifying relations. Instead of identifying attributes just by name, we will use both a name and relation. The above query is formulated in Haskell/DB as:

```

do{ x <- table students
  ; y <- table students
  ; restrict (x!mark ==. y!mark)
  ; restrict (x!name <.>. y!name)
  ; project (name = x!name, grade = x!grade)
}

```

Under the hood, we still generate relational algebra expressions but all the renaming is done automatically within the combinators. Besides automatic renaming, we would like the Haskell type-checker to prevent us from writing silly queries such as this one where we ask to project the non-existing city attribute of a student:

```

do{ x <- table students
  ; project (name = x!name, city = x!city)
}

```

We will present two designs for implementing comprehensions that are increasingly more type safe, but at the same time increasingly complex.

6.6 Attempt 1: Untyped comprehensions

The first attempt only hides the automatic renaming of attribute names, making this solution already much safer and convenient than writing abstract syntax directly. In our next attempt we will use phantom types to make queries more type safe. We defined the `Query` monad to express our queries. The use of a monad gives us the following advantages:

- The `do` notation provides a nice syntax to write queries (comprehensions).
- Monads enable a custom binding mechanism (ie `do{x <- table X; ...}`). to qualify names. An alternative approach of explicitly renaming attributes would be too cumbersome to use in practice.
- An invisible state can be maintained. The state of the `Query` monad contains the (partially) completed relational expression and a fresh name supply for automatic renaming of attributes.

Our query language now consists of three basic combinators, `restrict`, `project` and `table`, and the

two monad operations⁴ `returnQ` and `bindQ` for the `Query` monad. Besides that we have the usual binary combinators like `union`:

```

type State = (PrimQuery, FreshNames)
data Query a = Query (State -> (a, State))

```

```

returnQ :: a -> Query a
bindQ :: Query a -> (a -> Query b) -> Query b

```

```

restrict :: Expr Bool -> Query ()
project :: Rec r -> Query Rel
table :: Table -> Query Rel

```

```

union :: Query Rel -> Query Rel -> Query Rel

```

The exact details of doing correct renaming for all attributes are rather subtle and a thorough discussion is outside the scope of this paper. We will provide all the details in a separate report [13]. The `Rec r` and `Rel` types are explained in the next section where we add typed layer on top of the comprehension language.

6.7 Attempt 2: Typed comprehensions

We already know how to make the expression sub-language type safe using phantom types. For the comprehension language we will use the same trick. Central to this discussion is the attribute selection operator:

```

(!) :: Rel -> Attribute -> PrimExpr

```

Given a relation and an attribute name, the operator returns the attribute value expression. Given that any attribute always has a well defined type, we parametrize the attribute by its type to return an expression of the same type:

```

data Attr a = Attr Attribute

(!) :: Rel -> Attr a -> Expr a

```

Although we can now only use an attribute expression at its right type, the system doesn't prevent us from selecting non-existent attributes from the relation. The solution is to parametrize the `Rel` type by the its "scheme". Similarly, we parametrize the `Attr` type again by both the scheme of the relation and the type of the attribute:

⁴These functions allow us to use the `do` syntax.

```

data Rel r    = Rel Scheme
data Table r  = Table TableName Scheme
data Attr r a = Attr Attribute

```

The `Rel` and `Table` both retain their associated scheme. This is needed to read the concrete values returned by the actual query.

The selection operator `(!)` now expresses in its type that given a relation with scheme `r` that has an attribute of type `a`, it returns an expression of type `a`. The polymorphic types of the other basic combinators should not be too surprising:

```

(!) :: Rel r -> Attr r a -> Expr a

restrict :: Expr Bool -> Query (r)
project  :: Rec r -> Query (Rel r)
table    :: Table r -> Query (Rel r)

```

Our desire to guarantee type safety bears some additional cost on the user. For every attribute `attr` that occurs in a query, we have to define an attribute definition `attr :: r\attr => Attr (attr :: a | r)` a by hand, until TREX will provide first class labels. Similarly, for every base table with scheme `r` that we use, we need a definition of type `Table r`. For the example database of section 7 we have:

```

students :: Table(name :: String, mark :: Char)
name :: r\name =>
  Attr (name :: String | r) String
mark :: r\mark =>
  Attr (mark :: Char | r) Char

```

We have written a tool called `DB/Direct` that queries the system tables and automatically generates a suitable database definition. This tool is of course written using `Haskell/DB`.

The Haskell type-checker now checks the consistency of our queries. It accepts query passed without problems, but it fails to type check query `failed` because the condition `student!mark .<=. constant 5` wrongly attempts to compare a character to an integer and because the programmer accidentally used the attribute `ame` instead of `name`:

```

passed :: Query (Rel (name :: String))
passed =
  do{ student <- table students
    ; restrict (student!mark .>=. constant 'B')
    ; project (name = student!name)
  }

```

```

}

failed :: Query (Rel (name :: String))
failed =
  do{ student <- table student
    ; restrict (student!mark .<=. constant 5)
    ; project (name = student!ame)
  }

```

7 Exam marks

Any commercial exploitation of the web today uses server-side scripts that connect to a database and deliver an HTML page composed from dynamic data obtained from querying the database using information in the client's request. The following example is a simple server-side web script that generates an HTML page for a database of exam marks and student names.

The database is accessed via simple web page with a text entry and a submit button:

My name is:

The underlying HTML has a form element that submits the query to the `getMark` script on the server.

```

<HTML>
<HEAD> <TITLE>Find my mark</TITLE> </HEAD>
<BODY>
  <FORM ACTION="getMark.asp" METHOD="post">
    My name is:
    <INPUT TYPE="text" NAME="name">
    <INPUT TYPE="submit" VALUE="Show my mark">
  </FORM>
</BODY>
</HTML>

```

7.1 Visual Basic

Even the simplest Visual Basic solution uses no less than four different languages. Visual Basic for the business logic and glue, SQL for the query, and HTML with ASP directives to generate the result page.

1. In ASP pages, scripts are separated from the rest of the document by `<%` and `%>` tags. The prelude script declares all variables, constructs the query and retrieves the results from the students database. The ASP Request object contains the information passed by the client to the server. The Form collection contains all the form-variables passed using a POST query. Hence `Request.Form("name")` returns the value that the user typed into the name textfield of the above HTML page.

```
<%
Q = "SELECT student.name, student.mark"
Q = Q & " FROM Students AS student"
Q = Q & " WHERE \"student.name = \"
Q = Q & Request.Form("name")

Set RS = CreateObject("ADO.Recordset")
RS.Open Q "CS101"
%>
```

2. The body contains the actual HTML that is returned to the client, with a table containing the student's name and mark. The `<%=` and `%>` tags enclose Visual Basic expressions that are included in the output text. Thus the snippet:

```
<TR>
  <TD><%=RS("name")%></TD>
  <TD><%=RS("mark")%></TD>
</TR>
```

creates a table row that contains the name and the mark of the student who made the request:

```
<HTML>
<HEAD> <TITLE>Marks</TITLE> </HEAD>
<BODY>
  <TABLE BORDER="1">
    <TR>
      <TH>Name</TH>
      <TH>Mark</TH>
    <TR>
      <%Do While Not RS.EOF%>
    <TR>
      <TD><%=RS("name")%></TD>
      <TD><%=RS("mark")%></TD>
    <TR>
      <%RS.MoveNext%>
    <%Loop%>
  </TABLE>
</BODY>
</HTML>
```

3. The clean-up phase disconnects the databases and releases the recordset:

```
<%
RS.Close
set RS = Nothing
%>
```

7.2 Haskell

The Haskell version of our example web page is more coherent than the Visual Basic version. Instead of four different languages, we need only need Haskell embedded in a minimal ASP page [14]:

```
<%@ LANGUAGE=HaskellScript %>
<%
module Main where

import Asp
import HtmlWizard

main :: IO ()
main = wrapper $ \request ->
  do{ name <- request # lookup "name"
      ; r <- runQuery (queryMark name) "CS101"
      ; return (markPage r)
  }
```

The function `queryMark` is the analog of code in the prelude part of the Visual Basic page, except here it is defined as a separate function parametrized on the name of the student:

```
type Student = Row(name :: String, mark :: Char)

queryMark :: String -> Query Student
queryMark = \n ->
  do{ student <- table students
      ; restrict (student!name ==. lift n)
      ; project
        ( name = student!name
          , mark = student!mark
        )
  }
```

Function `markPage` makes a nice HTML page from the result of performing the query:

```
markPage :: [ Student ]
markPage = \rs ->
  page "Marks"
  [ table
    ( headers = [ "Name", "Mark" ]
      , rows = [[r!name, r!mark:""]
                | r <- rs
    )
```

```

    ]
  )
]
%>

```

The Haskell program is more concise and more modular than the Visual Basic version. Functions `queryMark` and `markPage` can be tested separately, and perhaps even more importantly, we can easily reuse the complete program to run in a traditional CGI-based environment, by importing the `CGI` module instead of `Asp` (in a language such as Standard ML we would have parametrized over the server interface).

8 Status and conclusions

The main lesson of this paper is a new design principle for embedding domain specific languages where embedded programs are compiled on-the-fly and executed by submitting the target code to a server component for execution. We have shown how to embed SQL into Haskell using this principle, but there are numerous other possible application domains where embedded compilers are the implementation technology of choice; many Unix services are accessible using a completely text-based protocol over sockets.

Traditionally, domain abstractions are available as external libraries. For instance, the `JMAIL` component (available at the time of writing at <http://www.dimac.net/>) provides a plethora of methods to compose email messages, to show just a few:

```

disinterface ISMTPMail {
  VARIANT_BOOL Execute();
  void AddRecipient([in] BSTR Email);
  [propget] BSTR Sender();
  [propput] void Sender([in] BSTR rhs);
  [propget] BSTR Subject();
  [propput] void Subject([in] BSTR rhs);
  [propget] BSTR Body();
  [propput] void Body([in] BSTR rhs);
};

```

Instead of providing a whole bunch of methods to construct an email message in an imperative style, an alternative approach would be to have a raw (SMTP) mail server [16] component that accepts

email messages in the RFC822 format [4] directly together with a set of combinators to build email messages in a compositional style. Ultimately, these abstract email messages are “compiled” into raw strings that are submitted to the mail server, perhaps by piping into the appropriate telnet port.

Our ultimate goal for a Domain Specific Embedded Compiler is to provide hard compile-time guarantees for type safety and syntactical correctness of the generated target program. Syntactical correctness of target programs can be guaranteed by hiding the construction of programs behind abstract data types. Phantom types, polymorphic types whose type parameter is only used at compile-time but whose values never carry any value of the parameter type, are a very elegant mechanism to impose the Haskell type system on the embedded language.

Our final example shows how Domain Specific Embedded Compilers can make server-side web scripting more productive. Because we can leverage on the abstraction mechanisms of Haskell (higher-order functions, module system), compared to the VB solution, the Haskell program is of higher quality, and easier to change and maintain. The formulation of queries using the `do{}` notation and extensible records is rather neat, but the exact translation into SQL turned out to be rather subtle.

Both the Haskell/DB and the DB/Direct packages are available on the web at the URL <http://www.haskell.org/haskellDB>.

Acknowledgements

Thanks to Hans Philippi for brushing up our knowledge on databases, and to the DSL99 referees, Arjan van Yzendoorn and Jim Hook for their constructive remarks that helped to improve the presentation of our paper. Joe Armstrong’s talk on services as components at the Dagstuhl workshop on “Component Based Development Under Different Paradigms” provided much of the initial inspiration for this work.

References

- [1] William J. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick II, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley Computer Publishing, 1998.
- [2] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension Syntax. *ACM SIGMOD Record*, 23(1):87–96, March 1994.
- [3] Peter Buneman and Atsushi Ohori. Polymorphism and type inference in database programming. *ACM Transactions on Database Systems*, 21(1):30–76, March 1996.
- [4] David H. Crocker. Standard for the Format of Arpa Internet Text Messages. Technical Report RFC 822, 1982. <http://www.imc.org/rfc822>.
- [5] C.J. Date. *An Introduction to Database Systems (6th edition)*. Addison-Wesley, 1995.
- [6] S.O. Finne, D. Leijen, E. Meijer, and S.L. Peyton Jones. H/Direct: A Binary Foreign Language Interface for Haskell. In *ICFP'98*, 1998.
- [7] S.O. Finne, D. Leijen, E. Meijer, and S.L. Peyton Jones. Calling hell from heaven and heaven from hell. In *ICFP'99*, 1999.
- [8] B.R. Gaster and M.P. Jones. A Polymorphic Type System for Extensible Records and Variants. Technical Report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham, 1996.
- [9] Paul Hudak. Modular Domain Specific Languages and Tools. In *ICSR5*, 1998.
- [10] Simon Peyton Jones and John Hughes (eds). Report on the Language Haskell'98. Available online: <http://www.haskell.org/report>, February 1999.
- [11] S.L. Peyton Jones, E. Meijer, and D. Leijen. Scripting COM Components in Haskell. In *ICSR5*, 1998.
- [12] P. J. Landin. The next 700 programming languages. *CACM*, 9(3):157–164, March 1966.
- [13] D. Leijen and E. Meijer. Translating notation to SQL. 1999.
- [14] Erik Meijer. Server Side Scripting in Haskell. *Journal of Functional Programming*, accepted for publication.
- [15] S. L. Peyton Jones and Philip Wadler. Imperative functional programming. In *20'th ACM Symposium on Principles of Programming Languages*, Charlotte, North Carolina, January 1993.
- [16] Jonathan B. Postel. Simple Mail Transfer Protocol. Technical Report RFC 821, 1982. <http://www.imc.org/rfc821>.
- [17] Dale Rogerson. *Inside COM*. Microsoft Press, 1997.

Verischemelog: Verilog embedded in Scheme

James Jennings

Eric Beuscher

Department of Electrical Engineering and Computer Science
Tulane University, New Orleans, Louisiana (LA) USA 70118
{jennings|beuscher}@eecs.tulane.edu

Abstract

Verischemelog (pronounced with 5 syllables, *ver-uh-scheme-uh-log*) is a language and programming environment embedded in Scheme for designing digital electronic hardware systems and for controlling the simulation of these circuits. Simulation is performed by a separate program, often a commercial product. *Verischemelog* compiles to Verilog, an industry standard language accepted by several commercial and public domain simulators.

Because many design elements are easily parameterized, design engineers currently write scripts which generate hardware description code in Verilog. These scripts work by textual substitution, and are typically ad-hoc and quite limited. Preprocessors for Verilog, on the other hand, are hampered by their macro-expansion languages, which support few data types and lack procedures. *Verischemelog* obviates the need for scripts and preprocessors by providing a hardware description language with list-based syntax, and Scheme to manipulate it.

An interactive development environment gives early and specific feedback about errors, and structured access to the compiler and run-time environment provide a high degree of reconfigurability and extensibility of *Verischemelog*.

1 Introduction

1.1 The Verilog Language

In this paper we describe a language for digital hardware design called *Verischemelog*, which compiles to Verilog.¹ Currently undergoing standardization, Verilog is a popular input language to sophisticated simulators of digital electronic circuits. Com-

¹Verilog is a trademark of Cadence Design Systems, San Jose, CA, makers of the Verilog XL simulation system. Verilog and VHDL are popular standards.

mercial simulators which accept Verilog input are used heavily in industry. Verilog is actually two languages: one describes hardware, the configuration of logic gates, wires, and other components; the other controls the event-based simulator, specifying input signals to circuits and the printing of textual output. The former part of Verilog is truly a configuration language, describing a static structure, and is sometimes called Verilog Hardware Description Language, or Verilog HDL. The other part of Verilog is sometimes called its “behavioral language,” an apt name because it specifies the behavior of the simulation. Explicit in the behavioral language are the passage of time, the occurrence of events, and general computation as well. Figure 1 shows Verilog code for a half-adder, and behavioral code for testing it on one possible set of inputs. Verilog has a C-like syntax [KR88], both for hardware specification and for behavioral code. The semantics of the Verilog behavioral language are complicated by the need to explicitly allow simulation time to pass, and by the possibly ambiguous effects of statements which execute “at the same (simulation) time.” In this paper we focus on hardware specification instead of simulation control for two reasons: the *Verischemelog* behavioral constructs simply mirror those of Verilog; and it is the need to *automatically synthesize* hardware description code which led us to develop *Verischemelog*.

1.2 Limitations of Verilog

The biggest limitation of the Verilog hardware description language is the lack of a facility for *generating* hardware description code. Verilog has a small macro language, essentially based on textual substitution, which does not allow, e.g. iteration. Therefore, although many designs are parameterized, there are no facilities for writing procedures which, when executed, *generate* hardware descriptions. Common design elements in digital systems include finite state machines (which are parameterized by their transition table),

```
// Half adder with propagation delay 10

module half_adder (bit1, bit2, sum, carry);
input  bit1;
input  bit2;
output sum;
output carry;

and #10 anon1(carry, bit1, bit2);
xor #10 anon2(sum, bit1, bit2);

endmodule
```

(a) Verilog hardware description

```
;; Half adder with propagation delay 10

(defmodule half_adder

  (interface (input bit1 bit2)
              (output sum carry))

  (description "Half adder with delay 10")

  (and (10) (carry bit1 bit2))
  (xor (10) (sum bit1 bit2)))
```

(b) Verischemelog hardware description

```
// Tests half adder on 1 + 1

module test_half_adder;

reg    in1;
reg    in2;
wire   sum;
wire   cout;

half_adder      anon1(in1, in2, sum, cout);

initial
begin
  in1 = 1;
  in2 = 1;
  #10;
  $display(
    "In: %d + %d ==> Sum: %d  Carry: %d",
    in1, in2, sum, cout
  );
end
endmodule
```

(c) Verilog test program

```
(defmodule test_half_adder
  (interface)
  (description "Tests half adder on 1 + 1")

  (reg in1 in2)
  (wire sum cout)
  (half_adder (in1 in2 sum cout))

  (initial
    (set! in1 1)
    (set! in2 1)
    (delay 10)
    ($display
      "In: %d + %d ==> Sum: %d  Carry: %d"
      in1 in2 sum cout)))
```

(d) Verischemelog test program

Figure 1: A half-adder computes the sum and carry of two 1-bit inputs. A Verilog module implementing a half-adder is shown on the left (a). On the right (b) is the Verischemelog equivalent. (c) and (d) are test programs that direct the simulator to provide a high signal, 1, for each of the input signals and then to display the output signals after a delay of 10 simulation time units. When a hardware module is instantiated, the module name appears first (followed by a propagation delay for logic gates), then the name for this particular instantiation, and then a connection list. The latter is a list of wires which must match the module's interface. The Verilog convention for logic gates is that **the output signal is listed first**.

The Verilog `#n` syntax indicates a delay of n simulation time ticks, as do the Verischemelog delay form and the numeric parameter between the gate name and the connection list. Verilog requires instantiated modules to have individual names; in Verischemelog they are optional.

arithmetic operations (parameterized by the size in bits of operands and results), multiplexers (parameterized by the number of inputs and their size in bits), etc. Many web sites by and for designers who use Verilog promote the sharing of techniques for writing scripts which produce hardware description code.² One engineer writes, “I have written hundreds (well, maybe I exaggerate a bit) of Perl, awk, and cshell [sic] scripts for processing my verilog code and for synthesis” [unk98a].

We have examined many of the scripts available online. They vary widely in implementation language, programming style, and documentation. They have in common the ad-hoc nature of translators and compilers which were not written using traditional techniques such as lexical analysis, parsing, etc. To supplement the efforts of the script writers, a few programmers have contributed preprocessors which do operate on syntactic structures. These also vary widely, although one in particular, `vpp` [unk98b], has both a small sublanguage of C-like mathematical expressions and a small set of iteration constructs. These facilities notwithstanding, `vpp` is fundamentally limited by its translation language which lacks procedures and has only integer and floating point data types.

1.3 Design of Verischemelog

`Verischemelog` is a replacement for Verilog which is designed to alleviate the need for ad-hoc scripts and limited preprocessors. `Verischemelog` has the following properties:

- `Verischemelog` is embedded in Scheme [CR98] and has Scheme-like list-based syntax.
- Scheme is used as the “macro language” for generating hardware description and behavioral test code.
- Many errors are reported interactively, giving more specific and immediate feedback than, for example, the commercial Cadence Systems simulator.
- `Verischemelog` compiles to Verilog, an industry standard.
- `Verischemelog` easily interfaces to existing Verilog code.

²A search of Yahoo (yahoo.com) gives 52 Verilog sites, many of which contain scripts, tips, and commercial design tools. AltaVista (altavista.com) reports 48740 sites referring to Verilog.

- The programming environment includes project management facilities in Scheme.
- Quantitative descriptions of hardware modules, including gate count and maximum signal propagation delay, can be automatically computed. The calculation of these and other attributes can be programmed at the user level.
- The system is extensible by the user, who may add new operators or procedures to `Verischemelog`, or even reconfigure the code generator.

Languages like Perl are popular for writing scripts which generate Verilog hardware descriptions because they have a rich set of string processing functions. By using Scheme instead, and operating on list-based structures instead of strings, the `Verischemelog` user has an advantage over the Verilog user which is analogous to the advantage of Scheme/Lisp macros over those of C (as implemented by `cpp`). In other words, `Verischemelog` provides Scheme as a macro language. The transformation and generation of hardware description code in `Verischemelog` relies on the list-based syntax of `Verischemelog` and the facility with which Scheme manipulates list structures.

Below in Section 2 we illustrate some notable features of `Verischemelog` using brief examples, and in Section 3 we present some longer examples from two parameterized CPU designs. In Section 4 we show how the user can access the internal system to reconfigure the compiler and add custom project management tools. Section 5 discusses the implementation, and Section 6 summarizes some related work.

2 Using Verischemelog

`Verischemelog` was designed in part for instructional purposes, as an alternative to Verilog for a senior-level undergraduate course in Computer Architecture, and as an example of a domain-specific language with a compact and accessible implementation for students studying programming languages and compilers. The considerable efforts put into scripts for synthesizing Verilog code by practicing engineers indicates that it may be appropriate for industrial use as well. In either environment, academia or industry, we assume the user is familiar with the Verilog language, and many constructs in `Verischemelog` exploit that familiarity by simply mirroring Verilog constructs with Scheme-like syntax. We also assume that users will design large systems in a modular fashion, and that some

modules will be written in Verilog itself. Finally, although Verilog is for us the target language, we expect that Verischemelog output may at times be read by humans, for instance by another designer who does not use Verischemelog. For this reason we have provided a simple facility for commenting Verischemelog programs such that the comments are transferred to the output code upon compilation. The output code is also properly indented, for the same reason.

2.1 Key Concepts

We note that a *module* is a unit of code in Verilog which may contain hardware description, behavioral code, or a mixture of the two. Modules contain instantiations of other modules, with their dependencies forming a directed acyclic graph. For example, the half-adder modules in Figure 1 (a) and (b) each instantiate one and gate and one xor gate, both of which happen to be primitive modules. Verischemelog preserves the character of Verilog's modules, which are unrelated to any modules or packages one may find in Scheme.

Verilog hardware modules have *interfaces* which specify the input and output signals to the module, giving them internal names and types. An instantiation of a hardware module must provide wires of the proper size and type, connected in the proper order, which is specified by the module's interface. In this context, the size of a wire is the number of conductors it carries. For example, a small bus may carry 8 wires. Wire types include *input*, *output*, *ground*, and several others. Deciding whether two types match is usually straightforward.

The *simulator* is used to test hardware designs. It is not part of Verischemelog. The simulator is said to *execute* behavioral code which provides input signals to a hardware design and displays output such as the values of output (and internal) signals of the module. Naturally, the test program may change the input signals over time, and make choices about what to display and when. Verilog's behavioral language, in which test programs are written, is much like a general purpose procedural programming language in which computation consumes zero simulation time. Verischemelog provides list-based syntax for Verilog behavior constructs and checks for several categories of errors, but provides no new behavior language constructs.

2.2 Working Interactively

Development in Verilog follows the usual sequence of edit, compile, and execute.³ In Verischemelog one works interactively at the Scheme read-eval-print loop. Using the Emacs editor with a Verischemelog session running in an editor buffer makes the experience quite similar to developing in Lisp or Scheme.⁴ Of course, one is conscious of the differences in both striking and subtle ways. The *defmodule* form, which ultimately defines a Verilog module, and its attendant declarations of wires and gates clearly mark the domain. A more subtle difference between Verischemelog and Scheme appears in the form of restrictions on symbol names in Verischemelog, which conform to Verilog's C-like syntax, containing only the underscore character and alphanumerics. We chose to use this syntax rather than translate from the larger universe of Scheme symbols in order to allow humans and programs to easily match Verischemelog (source language) names with their Verilog (target language) counterparts, e.g. in messages produced by the simulator during execution.

Hardware development then follows this process:

1. Verischemelog hardware description and test code is developed interactively using the *defmodule* form which invokes the front-end of the Verischemelog compiler, but does not generate Verilog code. It is here that syntactic and other errors are reported, and interfaces between modules are checked for consistency.
2. When a module is to be tested using the simulator, it is compiled using the *compile* procedure which generates Verilog code for the target module as well as conditionally for any modules on which it depends.
3. The simulator is invoked on the generated Verilog code, using a script written by Verischemelog which enumerates the output files needed and contains any desired simulator options. In a Scheme which supports such operations, Verischemelog can run the simulator in batch mode in a child process, displaying its output, etc.

In the sections that follow we show how Verischemelog can be used to automatically synthe-

³Some simulators, like Cadence Systems Verilog XL, combine compilation and subsequent execution into one step by default.

⁴We do not address in this paper the possible "culture gap" which in theory might face engineers trained in C-like languages who are presented with Verischemelog.

size hardware description and behavioral test code. We begin with the most basic feature, escaping to Scheme.

2.3 Escaping to Scheme

First note that we distinguish *evaluation* of a `defmodule` form from *compilation*. To evaluate is to run the front end of the `Verischemelog` compiler, including the type checker; to compile is to perform code generation. The `defmodule` form is implemented as a Scheme macro which evaluates Scheme code embedded in the `defmodule` body, inlines the results, and finally evaluates the module definition. Embedded Scheme forms begin with one of the reserved keywords `scheme` or `scheme-splicing`, which can be abbreviated using `$` or `$$` respectively.

Both `scheme` and `scheme-splicing` behave as a Scheme `begin` form: each subform is evaluated in sequence, and the value of the last form is returned. These forms parallel Scheme's `unquote` and `unquote-splicing`. The value of a `scheme` form is inserted in place directly into the surrounding `defmodule` code. The value of a `scheme-splicing` form must be a list; the list is "spliced into" the surrounding code.

Both escape forms may be used to generate hardware description code. A more basic use is to retrieve a value. For example, the propagation delays for the gates in the half-adder of Figure 1 could be retrieved from the Scheme variable `*delay*` as follows:

```
(define *delay* 14)

(defmodule half_adder

  (interface (input bit1 bit2)
             (output sum carry))

  (description
   (string-append "Half adder with delay "
                  (number->string *delay*)))

  (and ((scheme *delay*)) (carry bit1 bit2))
  (xor ((scheme *delay*)) (sum bit1 bit2)))
```

Note that the syntax for gate instantiations requires the delay to be in parentheses. Therefore:

`((scheme *delay*)) => (14).`

The same parameter could be used in the corresponding behavioral program which tests this half-adder, thus ensuring their consistency.

Note also that `Verischemelog` evaluates the expression in the `description` form, which must return a string. The `description` form (and a similar `comment` form) exist only in `Verischemelog`, not `Verilog`, and so the automatic evaluation without use of an escape form is mellifluous.

2.4 Synthesizing Systems

A common situation calling for automatic generation of hardware description is that of parameterizing a design element, such as an adder, by the number of bits in the word size of a Central Processing Unit (CPU). In this case we are synthesizing an entire system (a CPU) which may have many variable parameters, one of which, the word size, parameterizes the adder that will be used in the Arithmetic and Logic Unit (ALU). We may proceed in one of two ways. We may simply use an escape form such as `scheme-splicing` to generate a series of full-adders (Figure 2) or we may define a Scheme procedure which contains a `defmodule` form. For illustration, we will show the latter, the code for which is in Figure 2. Figure 3 shows the Verilog code resulting from (`make-adder 3`).

These very small examples serve as useful illustrations, but the true utility of `Verischemelog` can only be seen when designing large systems, such as a CPU. A CPU may be parameterized by word size, instruction format, bus configuration, etc. During the design process, a component such as the ALU may be implemented in a general form, parameterized like the adder of Figure 2. (This may encourage code reuse, as alternate configurations are easily generated from one implementation.) Other components, such a micro-programmed control unit, may be designed in a fixed (not parameterized) form.

Even in a fixed form, however, a control unit is easier to generate in `Verischemelog` than in `Verilog`, because attributes can be represented symbolically in Scheme data structures. For example, a Scheme variable can hold a list of symbols representing all of the control signals in the CPU. The control unit module can be built separately from the CPU, and we can ensure their interfaces will match because `Verischemelog` can *generate* their interfaces using the control signal list. Also, control signals can then be referred to symbolically, by their names, with `Verischemelog` automatically mapping the names to indexed references into a control bus such as `controls[27]`.

Likewise, in a few lines of Scheme one can write a small assembler for generating memory images to

```

(define make-adder
  (lambda (wordsize)
    (let ((desc
            (string-append (number->string wordsize)
                           "-bit adder with carry out")))
      (full-adder
       (lambda (carryin a b sum carryout)
         '(full_adder (,carryin ,a ,b ,sum ,carryout))))))

  (defmodule adder
    (description desc)
    (interface (input ((scheme wordsize)) a b)
               (output ((scheme wordsize)) sum)
               (output carry))

    (supply0 ground)
    (wire ((scheme (- wordsize 1))) c)

    (scheme-splicing
     (iterate wordsize
      (lambda (i)
        (full-adder (if (zero? i) ; carry in:
                        'ground ; 0,
                        (: 'c (- i 1))) ; or c[i-1]
                     (: 'a i) ; input a[i]
                     (: 'b i) ; input b[i]
                     (: 'sum i) ; sum[i]
                     (if (= i (- wordsize 1)) ; carry out:
                         'carry ; module output,
                         (: 'c i)))) ; or c[i]
        ))))

    (defmodule full_adder ; two half-adders make a full-adder.
      (interface (input carry_in bit1 bit2)
                  (output sum carry_out))
      (wire temp_sum temp_carry1 temp_carry2)
      (half_adder half_1 (bit1 bit2 temp_sum temp_carry1)
                  half_2 (carry_in temp_sum sum temp_carry2))
      (or (10) carry_or (carry_out temp_carry2 temp_carry1)))

```

Figure 2: A Scheme procedure which generates Verischemelog code for a “ripple” adder which contains *wordsize* full-adders. The input signals to full-adder *i* are bit *i* of each input wire *a* and *b*, and the carry out wire of the previous adder, *c[i - 1]*. The carry in of the first full-adder is wired to 0 (ground), and the carry out of the last full-adder is an output wire of the adder module.

The `:` procedure takes a symbol (the name of a wire) and one or two numbers (indices) and generates an array-like reference. Multiple-conductor wires in Verilog are declared as arrays with their leftmost and rightmost bit indices given, as in `input [2:0] a`, which declares a three-conductor wire named *a* whose conductors are numbered from left to right: 2, 1, 0. The procedure `iterate` is like a procedural form of Lisp’s `dotimes`, e.g.

`(iterate 5 (lambda (x) x)) ⇒ '(0 1 2 3 4).`

```
// 3-bit adder with carry out

module adder (a, b, sum, carry);

input  [2:0] a;
input  [2:0] b;
output [2:0] sum;
output carry;

supply0 ground;
wire    [1:0] c;

full_adder anon1(ground, a[0], b[0], sum[0], c[0]);
full_adder anon2(c[0], a[1], b[1], sum[1], c[1]);
full_adder anon3(c[1], a[2], b[2], sum[2], carry);

endmodule
```

Figure 3: The Verilog output from (make-adder 3).

be used in testing CPU designs. In Section 3 below we describe two large projects, each a CPU design, in which these techniques were employed.

2.5 Evaluation Catches Errors

As mentioned above, `defmodule` forms are evaluated interactively. `Verischemelog` reports an error when any of the following occur:

- Syntax errors.
- Module interface does not match connection list (a form of type checking).
- The identifier `z` is used as a variable name.

Verilog allows the use of `z` as a variable, but it is also the name of a constant meaning “high impedance.” Consequently, `z` is a “write only” variable in Verilog because when it appears in an expression it refers to the constant value. The lack of a warning or error by Verilog trips up many novice users, and probably others as well.

`Verischemelog` reports several types of warnings:

- Wire used but not declared.
- Module is missing a description, has a null body, is instantiated but not (yet) defined, etc.

- Signal possibly used as feedback.

These warnings illustrate how a customizable development environment can be used to promote good design practices. For example, Verilog allows the use of undeclared wires, but `Verischemelog` generates a warning because often this is the result of a typographical error. Similarly, `Verischemelog` encourages the use of the `description` form, which generates a block comment in the header of the output file. Descriptions of defined modules can also be searched interactively, to aid in project management. Finally, sometimes the output signal of a module is used inside the module itself as the input to another device. Although it can be done intentionally, some of the time this is the result of mis-wiring. The warning helps detect those cases, and it can be suppressed when the use of feedback is intentional.

2.6 Precompiled Modules

`Verischemelog` allows designers to declare modules “precompiled.” The designer provides the module name, its interface, and the filename of the Verilog code which defines it. Although `Verischemelog` will compile code which instantiates unknown modules (and will warn about it), the ability to declare precompiled modules enables the same interface type checking that would occur if the instantiated modules had

```

(define (make-mux-n-to-1 n bits)
  "BITS is the word size of mux input, N is the max number of inputs"
  (let* ((name (lambda () (make-symbol 'mux_n 'to 1 'bits 'bit)))
        (log-n (ceil-log-n n 2))
        (expt-n (expt 2 log-n)))
    (make-decoder-m-control-bits log-n)
    (make-tri-state-n-bits bits)
    (defmodule (name)
      (description desc)
      (interface
        (output ((scheme bits)) result)
        (input ((scheme bits))
          (scheme-splicing (map-bits n (lambda (n) (make-symbol 'in n))))))
      (input ((scheme log-n)) control))
    (wire ((scheme expt-n)) ndecode_out)

    ($$ (cons (list (make-symbol 'decoder_log-n 'to expt-n)
                    (make-symbol 'd log-n 't expt-n)
                    (list 'ndecode_out 'control))
              (map-bits n
                (lambda (n)
                  (list (make-symbol 'tri_state_bits 'bits)
                        (make-symbol 'ts bits 'n)
                        (list 'result
                            (make-symbol 'in n)
                            (: 'ndecode_out n))))))))))

```

(a).

```

module mux_2to1_4bit (result, in0, in1, control);

output [3:0] result;
input [3:0] in0;
input [3:0] in1;
input control;

wire [1:0] ndecode_out;

decoder_1to2 d1t2(ndecode_out, control);

tri_state_4_bits ts4_0(result, in0, ndecode_out[0]);
tri_state_4_bits ts4_1(result, in1, ndecode_out[1]);

// delay of 16

endmodule

```

(b).

```

module mux_4to1_9bit (result, in0, in1, in2, in3, control);

output [8:0] result;
input [8:0] in0;
input [8:0] in1;
input [8:0] in2;
input [8:0] in3;
input [1:0] control;

wire [3:0] ndecode_out;

decoder_2to4 d2t4(ndecode_out, control);

tri_state_9_bits ts9_0(result, in0, ndecode_out[0]);
tri_state_9_bits ts9_1(result, in1, ndecode_out[1]);
tri_state_9_bits ts9_2(result, in2, ndecode_out[2]);
tri_state_9_bits ts9_3(result, in3, ndecode_out[3]);

// delay of 24

endmodule

```

(c).

```

module mux_16to1_7bit (result, in0, in1, in2, in3, in4, in5, \
in6, in7, in8, in9, in10, in11, in12, in13, in14, in15, control);

output [6:0] result;
input [6:0] in0;
input [6:0] in1;
input [6:0] in2;
input [6:0] in3;
input [6:0] in4;
input [6:0] in5;
input [6:0] in6;
input [6:0] in7;
input [6:0] in8;
input [6:0] in9;
input [6:0] in10;
input [6:0] in11;
input [6:0] in12;
input [6:0] in13;
input [6:0] in14;
input [6:0] in15;
input [3:0] control;

```

```

wire [15:0] ndecode_out;

decoder_4to16 d4t16(ndecode_out, control);

tri_state_7_bits ts7_0(result, in0, ndecode_out[0]);
tri_state_7_bits ts7_1(result, in1, ndecode_out[1]);
tri_state_7_bits ts7_2(result, in2, ndecode_out[2]);
tri_state_7_bits ts7_3(result, in3, ndecode_out[3]);
tri_state_7_bits ts7_4(result, in4, ndecode_out[4]);
tri_state_7_bits ts7_5(result, in5, ndecode_out[5]);
tri_state_7_bits ts7_6(result, in6, ndecode_out[6]);
tri_state_7_bits ts7_7(result, in7, ndecode_out[7]);
tri_state_7_bits ts7_8(result, in8, ndecode_out[8]);
tri_state_7_bits ts7_9(result, in9, ndecode_out[9]);
tri_state_7_bits ts7_10(result, in10, ndecode_out[10]);
tri_state_7_bits ts7_11(result, in11, ndecode_out[11]);
tri_state_7_bits ts7_12(result, in12, ndecode_out[12]);
tri_state_7_bits ts7_13(result, in13, ndecode_out[13]);
tri_state_7_bits ts7_14(result, in14, ndecode_out[14]);
tri_state_7_bits ts7_15(result, in15, ndecode_out[15]);

```

// delay of 72

endmodule

(d).

Figure 4: (a) shows a mux-generator written in Verischemelog. (b), (c), and (d) show examples of muxes generated by the mux-generator with the following arguments: (make-mux-n-to-1 2 4), (make-mux-n-to-1 4 9), (make-mux-n-to-1 16 7).

been defined in *Verischemelog*. The *defcompiled* form used to declare precompiled modules can even be generated automatically from Verilog source files.

Another important use of *Verischemelog* is for efficiency. When a project is large, modules not actively being modified can be declared precompiled so that only their interfaces are loaded into *Verischemelog*, saving memory and time. Naturally, *Verischemelog* can generate the appropriate *defcompiled* form for any module already defined, so that in subsequent sessions only the short *defcompiled* forms need to be loaded, instead of the much longer *defmodule* forms.

3 Examples

Tables 1 and 2 show the sizes of two hardware designs implemented in *Verischemelog*. Each is a complete CPU with memory (or memories) and a microprogrammed control unit. Each was designed from the outset to be scalable; consequently the same *Verischemelog* code was used to generate variations of each machine with different word sizes. To generate the machines shown in the tables, a single parameter (the data size) was changed, and the modules recompiled. To test each variation, a small assembler (included in the figures in the table) was used to produce a memory image (or images) containing a program and data. Also counted in the figures in the table are library procedures of general utility, such as *iterate*.

Table 1 reflects a silicon approximation to a Turing Machine, a small processor with separate data and instruction memories. The machine has four instructions, *right*, *left*, *jump*, and *halt*. The *jump* instruction performs an unconditional branch to an absolute address. The *right* and *left* instructions each take three operands: *match*, *replacement*, and *goto*. A special purpose register, *H*, contains a data memory address, initially 0. The instruction

```
right    a, b, 1
```

does nothing when the contents of data memory at address *H* does not match *a*. The next instruction is fetched. If *a* is in memory at *H*, however, this instruction writes *b* there, increments *H*, and branches to 1. The operation of *left* is analogous, but *H* is decremented.

The CPU of Table 2 has a more traditional design, with 16 general-purpose registers, an ALU supporting integer arithmetic and bitwise logical operations, three internal busses, and a RISC-style instruction set in which the ALU operates on data in registers and

writes its result to a register. The instruction set contains about two dozen instructions, and the control unit is microprogrammed.

The tables illustrate that these parameterized designs required a substantial amount of *Verischemelog* code, but an amount roughly of the same order of magnitude as the generated Verilog code which otherwise might have been written by hand. For the Turing Machine, a 7 bit data size lets us write programs which manipulate ASCII characters on the tape, and we have not compiled a larger model. The processor of Table 2, on the other hand, has been compiled and tested for the purposes of illustration on data sizes so large as to be (currently!) impractical to build in silicon. We consider this a reasonable test of the HDL synthesis capabilities of *Verischemelog*.

4 Extending Verischemelog

An important aspect of *Verischemelog* is the extent to which it be customized, reconfigured, and extended by the user. Some of the customizations are:

- Selective control over the display of warnings
- Setting of various working and output directories
- Definition of simulator options
- Block comments and headers for output files

An important reconfiguration feature consists of a set of customizable tables used by the code generator. For example, users can interactively modify a list of known Verilog behavioral procedures. New built-in procedures in a new release of the simulator can be added this way. Users can also tell *Verischemelog* about new unary and binary operators. Finally, *Verischemelog* maintains a table of translations of operators and procedures from *Verischemelog* to Verilog. This allows *Verischemelog* users to write = instead of ==, bitwise-not instead of ~, etc.

One extensibility feature is the *verbatim* form, which allows the user to insert arbitrary strings into the Verilog output. While clearly of limited utility, it does provide a method of writing user-level code which generates Verilog output. Of course, such code is actually generated at evaluation time rather than code generation time. A structured way to change the code generator, for example to add new language constructs, is under consideration.

With respect to hardware analysis and project management, however, *Verischemelog* provides sufficient

Table 1: Various measures of the size of the Turing Machine implementation. Data size refers to the number of bits per cell on the tape, which was implemented as a finite memory. Measurements of lines of code do not include comments or blank lines. The term *definitions* refers to top-level `define` and `defmodule` forms. HDL code includes all forms necessary to generate Verilog HDL, including Scheme procedures. Test code includes all behavioral programs for testing the system and individual components.

Data Size	Verischemelog source		Verilog output
	Lines of code	Definitions	Lines of code
1 bit	732 HDL, 398 test	54 HDL, 35 test	530
2 bits	(same)	(same)	540
7 bits	(same)	(same)	589

Table 2: Various measures of the size of the EBP CPU implementation. Data size refers to the number of bits in a machine word (the size of the data bus, registers, ALU operands, etc.). Address size in each case was 16 bits. Measurements of lines of code do not include comments or blank lines. The term *definitions* refers to top-level `define` and `defmodule` forms. HDL code includes all forms necessary to generate Verilog HDL, including Scheme procedures. Test code includes all behavioral programs for testing the system and individual components. The numbers for the HDL code includes such high level abstractions as automatic generation of the control unit's lookup table and micromemory based on a symbolic description of the control signals, an assembler for writing memory images, and calculations of simulated hardware delays for choosing an optimal clock speed.

Data Size	Verischemelog source		Verilog output
	Lines of code	Definitions	Lines of code
16 bits	3246 HDL, 964 test	240 HDL, 27 test	1397
32 bits			1853
64 bits			2549
128 bits			4085
256 bits			7229
512 bits			13373
Totals	3246 HDL, 964 test	240 HDL, 27 test	30486 lines, 6 processors

structured access to its internals for the user to easily write code to:

- Generate a different script for invoking the simulator.
- Generate representations of module dependencies.
- Calculate the number of primitive gates or particular user-defined modules instantiated in a given set of modules.
- Calculate arbitrary statistics on hardware descriptions.

5 Implementation Notes

`Verischemelog` was written in Kali Scheme [CJK95], a variant of Scheme48 [KR94]. Scheme48 is a compact and portable implementation of Scheme with a module system, record package, and hygienic macros. Scheme48's Unix interface was sufficient to allow `Verischemelog` to launch the Verilog XL simulator in another process, and to display its output. The source code for the `Verischemelog` run-time environment, compiler, all data structures, and an online help facility totals 247 forms written in 3282 lines.

Kali Scheme is a novel distributed implementation of Scheme. By using it to implement `Verischemelog`, distributed synthesis, compilation, or analysis of large systems is possible. Owing to the strength of Kali's design, these distributed extensions can easily be written entirely in user space, and quite simply as well.

6 Related Work

The original inspiration for `Verischemelog` was a programming language and environment named THEE [Woo93]. THEE allows Common Lisp programmers to generate C code in much the same way that `Verischemelog` allows designers to generate Verilog.

Evidence that Scheme might mix well with a traditionally low-level "down and dirty" task such as hardware design came in the form of the Envision system [SF97] which extends Scheme for computer vision. The Envision environment contains a new language, statically-typed and with different semantics from Scheme in other respects, but with Scheme's syntax. Programmers write Scheme programs with embedded calls to procedures in the new language.

Those procedures are transmitted to another process, the "co-processor," which interprets them. The embedded language, designed for image processing, is interpreted only during development. At any time code in the embedded language can be compiled to C and linked with the "co-processor" for subsequent high-performance execution when called from Scheme.

Finally, the structured access to the internals of `Verischemelog` was inspired by the notion of meta-object protocols [KdRB91].

7 Conclusion

With Scheme as a macro language, `Verischemelog` users can easily generate code which compiles to Verilog, making synthesis of digital hardware designs much more efficient than programming directly in the output language. By providing structured access to the compiler and run-time system, `Verischemelog` enables a great deal of customization, reconfiguration, and extension. By performing syntax and type checking interactively, users get specific and timely feedback about errors. A variety of warnings reinforce good coding style as well as point out possible problems. The ability to interface with existing Verilog code and to handle moderately large designs indicates that `Verischemelog` may be a practical tool for use in industry.

The most important aspect of `Verischemelog`, however, is the central idea that to allow users to effectively generate code, they should be given a language designed to manipulate that code. We used Scheme for that language, and designed the list-based syntax of `Verischemelog` to be manipulated by it.

Acknowledgments

The source code, grammar, examples (including the Turing Machine) and design documents of `Verischemelog` are available on the web from <http://www.eecs.tulane.edu/www/Jennings>.

This paper describes research done in the Department of Electrical Engineering and Computer Science at Tulane University.

References

- [CJK95] H. Cejtin, S. Jagannathan, and R. Kelsey. Higher-order distributed objects. *ACM Transactions on Programming Languages and Systems*, September 1995.

- [CR98] W. Clinger and J. Rees. Revised⁵ report on the algorithmic language scheme. *ACM Sigplan Notices*, 33(9):26–76, September 1998.
- [KdRB91] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
- [KR88] Brian Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 1988.
- [KR94] Richard Kelsey and Jonathan Rees. A tractable scheme implementation. *Journal of Lisp and Symbolic Computation*, 7:315–335, 1994.
- [SF97] Daniel E. Stevenson and Margaret M. Fleck. Programming language support for digitized images or, the monsters in the closet. In *Usenix conference on Domain-Specific Languages*, pages 271–284, 1997.
- [unk98a] Author unknown. Celia's verilog page. <http://www.teleport.com/~celiac/tools.html>, last verified Sat Oct 3 13:03:13 CDT 1998.
- [unk98b] Author unknown. Vpp, a verilog preprocessor. <http://www.sybarus.com/product.htm>, last verified Thu Oct 8 17:03:13 CDT 1998.
- [Woo93] John Woodfill. The programming environment and language thee (tm). *Unpublished source code*, ©1990, 1991, 1992, 1993.

Declarative Specification of Data-intensive Web sites

Mary Fernández*
AT&T Labs - Research
mff@research.att.com

Dan Suciū
AT&T Labs - Research
suciū@research.att.com

Igor Tatarinov
North Dakota State University
tatarino@prairie.NoDak.edu

Abstract

Integrated information systems are often realized as *data-intensive Web sites*, which integrate data from multiple data sources. We present a system, called STRUDEL, for specifying and generating data-intensive Web sites. STRUDEL separates the tasks of accessing and integrating a site's data sources, building its structure, and generating its HTML representation. STRUDEL's declarative query language, called StruQL, supports the first two tasks. Unlike ad-hoc database queries, a StruQL query is a software artifact that must be extensible and reusable. To support more modular and reusable site-definition queries, we extend StruQL with functions and describe how the new language, FunStruQL, better supports common site-engineering tasks, such as choosing a strategy for generating the site's pages dynamically and/or statically. To substantiate STRUDEL's benefits, we describe the re-engineering of a production Web site using FunStruQL and show that the new site is smaller, more reusable, and unlike the original site, can be analyzed and optimized.

1 Introduction

In large corporations, high-speed intranets and Web browsers have increased the demand for integrated information systems. Before intranets, access to geographically dispersed information systems was usually limited to those people who administered the systems locally. In this environment, *data integration*, the task of integrating information from multiple data sources, was difficult, if not impossible. An AT&T customer, for example, may have multiple accounts, e.g., long distance and wireless, stored in separate account-management systems. An integrated "view" of customers' accounts is vital to many business processes, such as targeting new services to appropriate customers. Because of their value to diverse groups, integrated information systems must be easily accessible and therefore, are

usually realized as Web sites. These systems, which we call *data-intensive Web sites*, integrate information from multiple data sources, often have complex structure, and present increasingly detailed views of data, from a summary perspective at a top-level page to a detailed perspective at a lower-level page.

In previous work [9], we argued that building data-intensive Web sites is a data-management problem, whose solution consists of three main programming tasks: *accessing and integrating* the data available in the site, *building* the site's structure, i.e., specifying the data in each page and the links between pages, and *generating* the HTML representation of pages. To better support these tasks, we developed the STRUDEL system, which applies concepts from database-management systems to Web-site creation and management. STRUDEL's key idea is separating the management of a Web site's data, the specifications of its structure, and the HTML representation of its pages. STRUDEL provides a declarative *query language*, StruQL, for specifying the content and the structure of a Web site, and a simple *template language*, for specifying a site's HTML representation. STRUDEL's query interpreter automatically derives the site from a StruQL query. STRUDEL has many benefits: explicit separation of the three programming tasks allows multiple versions of a site to be derived from one specification [9], and StruQL's semantics supports verification of integrity constraints on a site's structure [10].

In this paper, we argue that building data-intensive Web sites is also an important *software-engineering* problem, and that a site's implementation, like other valuable software systems, should be extensible, reusable, and optimizable. For example, it should be easy for a site developer to integrate new data sources into a site and to derive a new site from an existing one. It should also be possible for the site developer to optimize overall site performance, for example, by using page-access patterns culled from a Web server to drive static and/or dynamic generation of the site's pages, or by identifying au-

*Contact author: Mary Fernandez, AT&T Labs - Research, 180 Park Ave., Room E243 Florham Park, NJ 07932-0971, 973-360-8679, (FAX) 973-360-8077

tomatically pages that contain data from the same sources and by caching shared data when it is expensive to compute. Site reuse is greatly simplified if the implementation clearly separates the definition of the site's content, structure, and presentation. Both the site-generation and data-caching problems are examples of site optimizations and are orthogonal to the site's definition. For example, choosing a site-generation strategy is analogous to optimizing a program given an execution profile.

In current practice, most data-intensive Web sites are implemented by loosely related programs written in imperative scripting languages, such as Perl. Scripting languages are well-suited for "gluing" together other software components [15], which makes them popular for constructing Web sites. The scripts for many site implementations, however, interleave the code for data access and integration, page construction, and HTML generation. Even when these tasks are separated, it is difficult to infer automatically the semantics of the script code. Interleaving of these tasks limits reuse of any one component and prevents analysis of the site implementation as a cohesive unit. Finally, the site-generation and data-caching strategies are usually encoded explicitly in the implementation, making it difficult to experiment with alternative policies.

In this paper, we show that StruQL is more effective than general-purpose scripting languages for implementing data-intensive Web sites. StruQL is an example of a *declarative query language*, and although it is not Turing complete, it has been used to implement several Web sites¹. Unlike higher-level programming languages, declarative query languages (e.g., SQL, OQL) usually express short, ad-hoc queries. They lack features, such as functions and modules, that support development of large software systems, which must be modular, extensible, and reusable. STRUDEL's application requires both the declarativeness of query languages and the functional constructs of higher-level programming languages. The main contribution of this paper is the integration of these features in one language. In particular, we extend StruQL with *functions* to improve the modularity and re-usability of site definitions and with *forms* to support dynamically bound inputs. We describe how the new language, called FunStruQL, supports flexible site-generation strategies and how forms can be specified declaratively.

¹ We encourage the reader to visit the STRUDEL-generated sites at <http://www.research.att.com/~mff,~suciu> and <http://www.research.att.com/conf/sigmod99>.

We support these claims through a case study of an internal AT&T Web site that is used in a production setting. We compare the site's original implementation with its complete re-implementation in STRUDEL and show that the new implementation is much smaller, more reusable, and unlike the original site, can be analyzed and optimized.

1.1 Case Study

To motivate STRUDEL's design, we first describe an internal AT&T Web site, called "hightoll notifier" (HTN), which identifies business-customer accounts that appear to be high risk, i.e., ones whose bills may go unpaid. Statistically, customers that have a significant increase in their telephone usage over a short period of time are more likely to not pay their bills than those customers that have constant daily usage. Other high-risk indicators include the customer's credit record and their ability to pay previous bills on time.

The data in the HTN site must be current to within one day or even a few hours, so that account representatives can identify and contact high-risk accounts before the account further increases its usage or goes into arrears at billing time. Before the HTN site existed, account representatives might have waited several weeks before they had sufficient information to identify high-risk accounts. The HTN site is a tremendous success, because it provides in real time an integrated view of high-risk accounts.

HTN is a good example of a data-intensive Web site: it integrates data from multiple sources and allows the site user to "drill down" from the high-level, summary perspective to the low-level source data. HTN computes usage statistics on approximately 250 million phone calls daily and integrates information from several sources: phone-call records, long-term account information, and credit records. Of the 1.7 million business accounts tracked, approximately 6000 are identified as potential risks. The site has five levels: each subsequent level provides a more detailed view of the high-risk accounts. The *root page* allows an account representative to select the types of high-risk accounts to track, e.g., a particular market segment. The *hot list* page lists the set of accounts in the chosen segment and orders them by a risk metric. The hot-list page points to *account pages*, which displays a summary of an account's usage in textual and graphical form. A *report page* is accessible from several pages in the site and presents the account's risk metrics and al-

lows the account rep to view and record interactions with the customer. The most detailed page presents the original phone-call records from which the usage summary is computed.

The first HTN site was implemented using scripting languages, e.g., Korn shell and Perl, and common Unix command-line tools, e.g., awk, sed, and grep. The scripts process user inputs, invoke Unix tools to handle simple data-management tasks, and format and emit HTML pages. Several C programs implement rudimentary database operations. Although some scripts differentiated the three site-creation tasks, most scripts interleaved them. The result is a loosely related set of scripts that implement the required functionality, but that have the characteristics of a poorly implemented software system: the code is hard to understand and extend, because the program's tasks are undifferentiated. These problems complicated extension and prevented reuse of HTN's first implementation.

Given these limitations, the site was re-engineered using STRUDEL [9] and the Daytona relational database management system [13]. The short-term goal was for the new implementation to simplify maintenance and extension of the site. The long-term goal was to show that declarative specification supports site reuse and flexible site-generation strategies. Overall, the STRUDEL implementation is 1.6 times smaller than the original implementation, but if we compare only the code that defines the site's content and structure, i.e., the code that a developer must understand to reuse or extend the site, it is more than 4 times smaller. Section 6 presents an evaluation of this re-engineering effort.

2 STRUDEL Overview

We first describe STRUDEL's architecture, depicted in Figure 1, before focusing on its query language. Rectangles depict processes and emboldened terms specify the inputs and outputs of the processes.

STRUDEL supports two common types of Web-site data: *semistructured data* and *tuple-stream data* (bottom of Fig. 1). Semistructured data is characterized as having few type constraints, irregular structure, and rapidly evolving or loosely defined schema [1]. Web sites and XML data [7] are good examples of semistructured data. For example, XML elements can have missing attributes or attributes whose value is not strictly typed (e.g., a **name** attribute may have an atomic value in one element

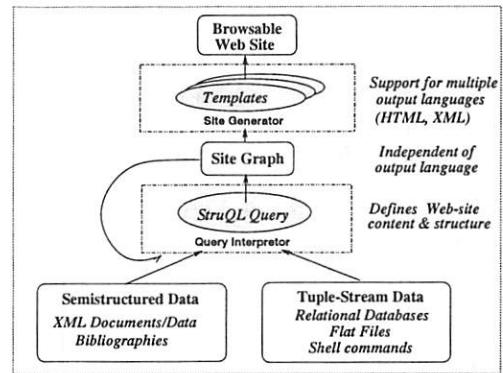


Figure 1: STRUDEL Architecture

and a complex value, (**lastname**, **firstname**), in another element.).

As in related systems [8], STRUDEL represents semistructured data as a *collection* of objects, in which each object is either *complex* or *atomic*. A complex object is a set of (*attribute*, *object*) pairs, and an atomic object is an atomic value (e.g., **int**, **string**, **video**). Hence, data is a *graph*, with edges labeled by attributes and leaves labeled with atomic values. Internal nodes have unique object identifiers, called *OIDS*, and data can be exchanged in a text representation. For example, Fig. 2 contains a fragment of an address database in XML. The complex objects **addresses** and **entry** have object identifiers (the **id** attribute). In semistructured data, the names, types, and cardinality of attributes may vary. For example, the first **entry** element has two **street** attributes, but the second has only one; the second has a **postalcode** attribute, but the first has a **zip** attribute. By default, the root XML node (e.g., the **addresses** element) is contained in the STRUDEL collection **XMLRoot**. We use the terms nodes and objects interchangeably, but note that object does not denote a strictly typed value as it does in an object-oriented language or database.

STRUDEL was initially designed to query and manage only semistructured data. Most Web sites, however, integrate information from well-structured data sources, such as relational databases, flat files, or the output of ad-hoc shell commands. STRUDEL, therefore, also supports tuple-stream data, i.e., any data source that can be modeled by a finite stream of fixed-width records.

A StruQL query (middle of Fig. 1) is applied to semistructured and/or tuple-stream data sources, and its result is a *site graph*, which represents the site's content and structure and is completely in-

```

<!-- Postal addresses in XML -->
<addresses id="alladdrs">
  <entry id="addr1">
    <name>AT&T Research</name>
    <address>
      <street>180 Park Ave.</street>
      <street>Bldg. 103</street>
      <city>Florham Park</city>
      <state>NJ</state>
      <zip>07932</zip>
    </address>
  </entry>
  <entry id="addr2">
    <name>INRIA Rodin</name>
    <address>
      <street>BP.105 Rocquencourt</street>
      <city>Le Chesnay</city>
      <postalcode>cedex 78153</postalcode>
      <country>France</country>
    </address>
  </entry>
</addresses>

```

Figure 2: Example of semistructured data

dependent of the output language. StruQL queries are compositional: a site graph is another example of semistructured data and can be queried by another StruQL query. A site graph is externalized on disk as an XML document.

To produce a browsable Web site, an *HTML template* is associated with each object in the site graph. Objects in a site graph may represent complete pages or page components. Usually, a template is associated with a collection of related objects, e.g., all account-page nodes. A template interleaves plain HTML text with STRUDEL-specific tagged expressions that access an object's attributes and format attributes' values. The template language is similar to other languages that separate presentation from content [5]. This technique simplifies the site programmer's task: he writes plain HTML extended with simple programmatic constructs, instead of a more complex scripting program that generates HTML. STRUDEL's generator (top of Fig. 1) evaluates the appropriate template for every object in a site graph. The resulting pages are the browsable Web site. Section 5 describes the template language in more detail.

2.1 Related Systems

Many commercial systems exist for designing and implementing Web sites. They include WYSI-

WYG HTML editors, tools for integrating database queries in individual Web pages, and model-driven, Web-site design systems. We focus on research systems and refer the reader to thorough reviews of site-development tools [11, 12]. Most of the problems associated with designing a Web site, such as modeling the site's content, specifying navigational structure, and customizing visual presentation, have been studied in the context of hypermedia systems, and many of the solutions for hypermedia systems are transferrable to Web-site design. The Autoweb [16], OOHDM [17], and Araneus [6] systems ascribe to a formal methodology of Web-site design, whose purpose is to isolate the various tasks of site design. Each system provides different tools, with varying levels of automation, to implement a design. Because their primary purpose is site design, neither Autoweb nor OOHDM-Web support querying or data integration. Like STRUDEL, Araneus separates data integration, site definition, and visual presentation, but it has two data models (one relational and one strictly typed graph) and two query languages, which cannot be composed naturally. We note that as an implementation tool, STRUDEL is complementary to site-design tools, because StruQL is well-suited to automatic generation and could be used as an implementation language for a variety of design systems.

Mawl [5] is a device-independent language for programming form-based services, which can be realized as Web applications or as interactive voice-response systems. Although STRUDEL's application is different, its separation of application logic from presentation and its template language are both inspired by Mawl.

The Document Object Model (DOM) [3] is a language-independent API for accessing HTML and XML documents, and the Extensible Stylesheet Language (XSL) [14] is a rule-based language for rendering a document in a markup language. These emerging standards are document centric, but may influence STRUDEL; e.g., a site graph could implement the DOM interface and possibly be rendered using XSL instead of STRUDEL's template language.

3 StruQL Query Language

We describe StruQL's core syntax by example, give an informal semantics, and describe query evaluation. In Sec. 4, we extend the StruQL with functions and forms. We illustrate StruQL's features

```

1  // Link root page to page of all accounts
2  link Root() -> "Accounts" -> AccountsPage()
3  // AccountsPage refers to each account in account database and its associated page
4  { where (acct, name, street, city, state, zip) in SQL.query("AccountDB", "select acct ...")
5    link AccountsPage() -> "Info" -> Info(acct),
6      Info(acct) -> { "Acct" acct, "Name" name, "Street" street,
7                    "City" city, "State" state, "Zip" zip,
8                    "AcctPage" AcctPage(acct) },
9    AcctPage(acct) -> "Info" -> Info(acct)
10
11  // AcctPage refers to non-zero usage records in the usage database.
12  { where (date, dom is int, intl is int) in SQL.query("UsageDB", "select date ...", acct)
13    dom + intl > 0
14    link AcctPage(acct) -> "UsageData" -> UsageData(acct),
15          UsageData(acct) -> "Entry" -> UsageEntry(acct, date),
16          UsageEntry(acct, date) -> { "Date" date, "Total" (dom + intl) }
17  }
18  // Query postal database to determine possible aliases for account
19  { where XMLRoot{root}, root -> "addresses"."entry" -> addr,
20        addr -> { "name" alias, "address"."street" street1, "address"."zip" zip },
21        street1 = street
22    link Info(acct) -> "Alias" -> alias
23  }
24 }

```

Figure 3: Fragment of site-definition query for `AcctPage` in HTN site

using the query in Fig. 3, which defines the account page in the HTN site. The site's data sources are two relational databases, of accounts and phone-call records, and one semistructured source of addresses. We focus on StruQL's declarativeness, i.e., a query specifies *what* the site's content and structure is, not *how* it is computed; its support for multiple data sources; and its controlled use of a general-purpose programming language (Java).

A StruQL query is a function that maps input-graph nodes and atomic values to a graph. A query's body is defined by the first EBNF grammar rule² in Fig. 4. The *where* clause is a conjunctive predicate expression. The link expressions link new nodes in the site graph, and collect expressions put nodes in the site graph's collections. *Node constructors* denote the OIDs of new nodes in the site graph. A predicate is a *collection expression*, a *regular-path expression*, an *atomic predicate*, or an *external-source expression*.

The query in Fig. 3 illustrates most of StruQL's features. The first clause on line 2 has an empty *where* clause, which is always true, so its associated link expression is always evaluated. `Root` is a node constructor that creates new object OIDs. By definition, a node constructor when applied to

the same tuple of values always produces the same OID, so this expression creates two unique nodes, named `Root()` and `AccountsPage()`, and a link labeled `"Accounts"` between them.

The second clause (lines 4–9) contains an *external-source expression*. This expression binds the variables `acct`, `name`, etc. to the stream of tuples produced by the SQL query applied to the accounts database, `AccountDB`. For each binding of `acct`, the link expression on line 5 creates a new object, `Info(acct)`, and links `AccountsPage` to it. The expression on lines 6–8 copies all of `acct`'s attributes and values into the new `Info` object and links `Info` to its associated `AcctPage` object. Cycles between objects are permitted: line 9 links `AcctPage(acct)` back to its associated `Info` object. The nested clause (lines 12–17) is similar; it queries the usage database, which produces the domestic and international phone-call usage records for each `acct`. The *where* clause is satisfied when the sum of `dom` and `intl` is non-zero. The associated link clause groups usage entries by date in `UsageEntry(date, acct)` and stores the sum of `dom` and `intl`.

The last clause (lines 18–22) contains a *collection expression*, which binds a variable to every node in the specified collection, and *regular-path expressions*, which match arbitrary paths in an input

² `{ }` delimit sequences, `|` separate alternatives, and `[]` delimit optional constructs

$Body : - [where \{ Predicate \}]$	$Predicate : - CollectionName\{Var\}$
$[link \{ NodeConstructor \rightarrow AttrExpr \rightarrow Term \}]$	$ Var \rightarrow RegularPathExpr \rightarrow Term$
$[collect \{ CollectionName\{NodeConstructor\} \}]$	$ AtomicPredicate$
$\{ \{ Body \} \}$	$ (\{ Var \}) in ExtSourceExpr$

Figure 4: *StruQL*'s EBNF grammar rules.

graph, e.g., an XML document. This clause is satisfied when there exists an object `addr` that is reachable from any member of the `XMLRoot` collection by a sequence of edges labeled "addresses"."entry"; `addr` must also have outgoing edges labeled `name` `address.street`, and `address.zip`. In general, a condition of the form $x \rightarrow R \rightarrow y$ means that there exists a path from node x to node y that matches the regular-path expression R . In addition to the concatenation (.) operator, R may contain the alternation (|) and (*) Kleene star operators.

The `where` clause on lines 19–22 is only satisfied when the values of `addr`'s `address.street` and `address.zip` attributes equal the values of `street` and `zip` bound in the first clause. In database parlance, this expression is a *join* on `street` and `street1`, or in logic-programming parlance, `street` and `street1` are unified. The two interpretations are equivalent. An explicit condition is not necessary: the join on `zip` is implicit, because it appears as the target of `address.zip`.

Figure 5 depicts a fragment of the site graph produced by the query in Fig. 3 applied to the sample relational data in Fig. 5 and the XML data in Fig. 2. The graph encodes the site's content and structure; e.g., the `Info` objects have links to account names and to account pages. The choice to realize objects as pages or as page components is delayed until HTML generation; our choice of node-constructor names (e.g., `AcctPage`) hints that some objects will be realized as pages, but this is not a requirement.

StruQL accesses user-defined methods using the Java reflection mechanism [4]. Any static Java class that implements *StruQL*'s predicate, expression, or external-source interfaces is permissible. For example, *StruQL* provides the package `SQL` for accessing JDBC-compliant databases; it also provides a tuple-stream interface for flat files and shell commands and an interface to a Perl library for regular expres-

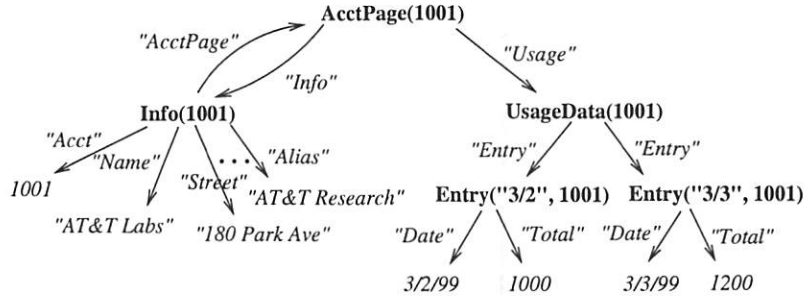
sion string matching.

StruQL supports the atomic types integer, float, string, date, and mime-content types such as URL, image, html, and postscript. By default, all values are interpreted as strings, but any variable can be annotated with an optional type, such as the `dom` and `int1` variables in Fig. 3(line 12). The query interpreter attempts to coerce values to the appropriate type at runtime. Although static typing is preferable, dynamic typing is necessary for *StruQL*, because the types of atomic values in data sources are usually unknown until query-evaluation time.

3.1 Semantics and Query Evaluation

A *StruQL* `where` clause is a *conjunctive query*. Conjunctive queries [2] are an important class of database queries, because they have several desirable properties. First, whenever the domain of a conjunctive query is finite, its range is also finite, which means that an evaluation of the query is guaranteed to terminate. Second, the *query equivalence* and *query containment* problems are decidable for conjunctive queries. Formally, given two queries Q_1 and Q_2 , query equivalence decides for all inputs I , $Q_1(I) = Q_2(I)$, i.e., Q_1 and Q_2 compute the same result; query containment decides for all I , $Q_1(I) \subseteq Q_2(I)$. Query optimizers may rely on query equivalences and containments to eliminate redundant computations.

StruQL has an *active-domain* semantics, which means that the domain of a query, \mathcal{D} , is the union of the finite domains of the query's input graphs (i.e., object OIDs and atomic values), of its external sources, and of the set of constants that occur in the query. *StruQL*'s semantics can be described informally in two stages. The *query stage* depends only on the `where` clause and produces all possible bindings of variables to values in \mathcal{D} that satisfy all conditions in the clause. Its result is a relation R with one attribute for each variable; each tuple t in R satisfies the conditions. The *construction stage*



Sample Data:

AccountDB() \ni (1001, "AT&T Labs", "180 Park Ave", "Florham Park", NJ, 07932)

UsageDB(1001) = {(3/2/99, 1000, 100), (3/3/99, 1200, 50)}

Figure 5: Fragment of site graph generated by query in Fig. 3

constructs the site graph by evaluating each link and collect expression once for every tuple t in R . The node constructor $N(v_1 \dots v_n)$ denotes the object in the site graph whose OID is the value $N(\pi_{v_1 \dots v_n}(t))$, i.e., the value of variables $v_1 \dots v_n$ in t . Each link expression, $N(v) \rightarrow A \rightarrow N(w)$ creates an edge labeled A from $N(\pi_v(t))$ to the node denoted by $N(\pi_w(t))$. Each collect expression $C\{N(v)\}$ adds the node $N(\pi_v(t))$ to the collection C .

A StruQL query is evaluated by interpreting a *physical-operation tree*. STRUDEL's query-plan generator, like traditional query processors, translates a StruQL query into a physical-operation tree. Query-plan generation is similar to code generation in a compiler. The details of query-plan generation and strategies for efficient evaluation and optimization of StruQL are described elsewhere [9].

An important implication of StruQL's semantics is that a site graph is finite. In STRUDEL's first implementation, queries were evaluated completely and therefore materialized the entire site graph. We call this an *eager* evaluation strategy, which produces a *static* site graph. Eager evaluation is not always feasible or appropriate. For example, the query may range over gigabyte-sized data sources and therefore produce very large site graphs, or the site may depend on dynamically bound variables, e.g., inputs derived from a form. In addition to eager evaluation, STRUDEL now supports *lazy* evaluation, in which part of a site query is evaluated dynamically, e.g., at "click time"; a lazily evaluated query produces a *dynamic* site graph. Next, we introduce StruQL's functions, which modularize site-definition queries and are the minimal unit of query evaluation, and forms, which support dynamic binding of variables. After describing their semantics, we describe how flexible site-generation strategies can

be implemented by combining eager and lazy evaluation of functions to produce sites that have both static and dynamic parts.

4 Extending StruQL with Functions

Our first applications of STRUDEL were small Web sites, like personal home pages, designed and maintained by one person. The HTN site was STRUDEL's first production application, and its site definition must be understandable and possibly extended by multiple people. HTN has several types of pages and several data sources, which made its definition in StruQL long and unwieldy.

To address these issues, we extended StruQL with functions. Functions are unusual in query languages³, and adding them to StruQL is novel. StruQL's functions also differ from functions in general-purpose programming languages.

A StruQL function maps values in \mathcal{D} (i.e., atomic values and input object OIDs) to a unique object (node) in the site graph and defines the entire subgraph accessible from that object. A function is defined by the EBNF grammar rule:

Function : – fun *ID* ({*Var*}){*Body*}

A function has a name (*ID*) and formal arguments (*Var*'s), and its body consists of a StruQL query. The meaning of a function is that it constructs a subgraph and returns a reference to the graph's root. In the function body, the reserved node constructor **Result()** denotes the subgraph's root. There are two kinds of function calls, eager (**!f(x,y,...)**) and lazy (**?f(x,y,...)**), which we describe below. A *function call* is like a a

³While SQL defines *stored procedures*, these are not queries per se, but form a different language.

node constructor, i.e., it can occur in a collect expression, or as the target of a link expression:
`link AcctEntry(acct) -> "AcctPage" ->`

`?acctPage(acct) // line 12 in Fig. 6`

Here, `AcctEntry` is a node constructor and `acctPage` is a function call.

The subgraph returned by a function is disjoint from the graph constructed by the rest of the query and is connected to the rest only by edges to its root; e.g., the link expression above, constructs an edge "AcctPage" to the root of the subgraph returned by `acctPage(acct)`. Node constructors are locally scoped in a function's body, which guarantees that the nodes it constructs are disjoint from all others. For example, the body of the function `acctPage` in Fig. 6 contains the node constructors `Result`, `UsageData`, and `UsageEntry`, and these names are local to the function `acctPage`. Otherwise, function calls behave like node constructors, i.e., multiple calls to `acctPage(a)` with the same value for `a` produces exactly the same subgraph.

The `!()` prefix is a function-evaluation directive that specifies a callee function should be evaluated eagerly (lazily) when its caller function is evaluated. In Fig. 6, `acctInfo` is always evaluated eagerly; `reportPage` is evaluated lazily when called from `hotList`, but eagerly when called from `acctPage`. The user chooses a lazy or eager strategy based on efficiency considerations; the strategies' semantics are equivalent, i.e., both produce the same graph. We plan to generate strategies automatically and are experimenting with an engine that treats directives as "hints" that can be overridden. For example, an alternative strategy might evaluate all calls to `reportPage` lazily, because the page is accessed infrequently or because it is expensive to compute. In Sec. 4.2, we describe how one query can be evaluated using different site-generation strategies.

When a function is evaluated, a call to a lazily evaluated callee is replaced by a *closure* node, which encapsulates the information necessary to evaluate the callee. Calls to eagerly evaluated callees are replaced by the callee's result node. The HTML generator (Sec. 5) emits the appropriate HTML for either case.

Given FunStruQL's declarative semantics, functions differ fundamentally from those in other programming languages. All FunStruQL functions can be inlined, while preserving the query's semantics; in programming languages like C++ or ML, only non-recursive functions can be inlined. For example, the

FunStruQL function:

```
fun f(x) { link Result() -> "self" -> !f(x) }
```

returns a node with a link to itself and is guaranteed to terminate. A call to `f`:

```
link Node(y) -> "call" -> !f(z)
```

would be inlined as:

```
link Node(y) -> "call" -> f_Result(z),  
    f_Result(z) -> "self" -> f_Result(z)
```

Of course, inlining does not preserve the *operational* semantics of lazy functions, so our interpreter does not inline lazy function calls. In FunStruQL, function call arguments may be variables and constants, but not arbitrary expressions; this prevents an eager evaluation from resulting in a non-terminating computation, as it would in:

```
fun f(x) { link Result() -> "self" -> !f(x+1) }
```

Figure 6 contains the query for the HTN site. The function `hotList` (line 8) defines the top-level page that contains a list of those accounts in the `HotList` database that also occur in the account database, `AccountDB`. The wildcard `_` ignores the value produced by an external source; in this case, `acct` must have *some* value in `AccountDB`, but the value itself is ignored. For each such account, a link is created to the corresponding `reportPage`, `acctPage`, and `acctInfo` nodes. The function `acctInfo` (line 17) computes general account information that is shared among several nodes and appears on multiple pages in the site. The `acctPage` function (line 23) links its result to the corresponding `reportPage` and accesses all the non-zero usage records from the `UsageDB`. It groups them by `date` and sums the `dom` (domestic) and `intl` (international) usage values. The `reportPage` function (line 33) lists all the reports known for the given account by querying an external reports database.

From a software-engineering perspective, this query has several desirable properties. Each function *identifies* the sources on which it depends, which makes it possible to reuse and modify the definition easily. A function also *encapsulates* a page, several pages, or a page component, making it possible to reuse parts of the site in different contexts; e.g., `acctInfo` is a page component contained in `acctPage` and `reportPage`.

```

1  // Root contains form to select hotList and one to select a specific account
2  { where accttype in { "SmallBiz", "MiddleMarket", "ISP" }
3    link Root() -> { (age, type) from HotListForm -> ?hotList(type, age),
4                      (acct) from AcctForm -> ?acctPage(acct) },
5    Root() -> "AcctType" -> accttype
6  }
7  // hotList lists those accounts in HotList database that also occur in Account database
8  fun hotList(type, age) {
9    where (acct, rank) IN HotList(type, age), (_, _, _, _, _) IN AccountDB(acct)
10   link Result() -> "Entry" -> AcctEntry(acct),
11         AcctEntry(acct) -> { "ReportPage" ?reportPage(acct),
12                             "AcctPage" ?acctPage(acct),
13                             "Info" !acctInfo(acct),
14                             "Rank" rank }
15  }
16  // General account information is used in hotList, acctPage and reportPage
17  fun acctInfo(acct) {
18    where (name, street, city, state, zip) IN AccountDB(acct)
19    link Result() -> { "Acct" acct, "Name" name, "Street" street,
20                     "City" city, "State" state, "Zip" zip, "AcctPage" ?acctPage(acct) } }
21  // acctPage links to the acct's reportPage and to non-zero usage records
22  // in the usage database.
23  fun acctPage(acct) {
24    link Result() -> { "Info" !acctInfo(acct),
25                     "ReportPage" !reportPage(acct) }
26    { where (date, dom, intl) IN UsageDB(acct), dom + intl > 0
27      link Result() -> "UsageData" -> UsageData(),
28            UsageData() -> "Entry" -> UsageEntry(date),
29            UsageEntry(date) -> { "Date" date, "Total" (dom + intl) } }
30  }
31  // reportPage lists all the reports known for the account by querying
32  // an external reports database
33  fun reportPage(acct) {
34    link Result() -> "Info" -> !acctInfo(acct),
35    { where (exec, date, comments) IN ReportsDB(acct)
36      link Result() -> "Entry" -> ReportEntry(exec, date)
37            ReportEntry(exec, date) -> { "AcctExec" exec,
38                                         "Date" date,
39                                         "Comments" comments } }
40  }

```

Figure 6: Site-definition query for HTN site

4.1 Forms

Forms are an important feature of Web sites that cannot be expressed easily in a declarative query language, because they model sequential operations: get values from the user, then consume the values by constructing new graph nodes (i.e., pages). A FunStruQL *form* has the syntax:

Form : - (*{Var}*) from *ID*

A form may only occur on an edge in a link expression. Its effect is that, on traversing that edge, the user is prompted for the form variables' values, then the destination node is constructed; the latter must be a lazily evaluated function, because its arguments are not bound until runtime. For example:

```
link Root() -> (acct) from AcctForm ->
    ?acctPage(acct) // Fig. 6, line 4
```

defines the form `AcctForm` in the `Root` node. The HTML page for `Root()` must contain a form that binds `AcctForm`'s variables; this requirement is enforced by the HTML generator. Submission of the form's inputs by the user corresponds to a traversal of the edge `AcctForm` in the site graph; this binds the variable `acct` to the input value, and then the function `acctPage(acct)` is evaluated.

Conceptually, a query with forms defines an infinite graph, because the form's range of values can be infinite. This is not a problem, however, because only a finite portion of the site graph is ever expanded. Even though FunStruQL cannot restrict the set of values produced by a form to \mathcal{D} , the query's domain, we can check declaratively the validity of user's inputs. For example, the `type` of the `HotListForm` on line 3 should be restricted to one of the `AcctType` values. We could enforce that restriction in `hotList`:

```
fun hotList(type, age) {
  {where type in {"SmallBiz",
                 "MiddleMarket", "ISP"},
   (acct, rank) IN HotList(type, age),
   (_, _, _, _, _) IN AccountDB(acct)}
  link // . . . from hotList in Fig. 5
}
{ // Error clause
  where not(type in {"SmallBiz",
                    "MiddleMarket", "ISP"})
  collect ErrorPage{Result()}
  link Result() -> "BadArgument" -> "type",
        Result() -> "BadValue" -> type
}
```

The second `where` clause produces a node in the `ErrorPage` collection, which when realized in HTML, reports the invalid input to the user. It would be useful to have these declarative error clauses generated automatically given the range of an input variable.

4.2 Site-generation Strategies

The ability to support multiple site-generation strategies is especially important for data-intensive Web sites, in which the time to produce pages is non-uniform; e.g., some functions may submit expensive queries to an external source. In current practice, however, it is difficult for a site developer to support more than one site-generation strategy. So by default, most sites are generated either dynamically or statically.

STRUDEL already supports site-generation strategies defined explicitly in the query, but we would also like to support those defined automatically by a profile-driven site optimizer. For example, when account reps begin work each day, they scan the hot lists for new accounts in their market segments. The first pages accessed in the site can be identified by examining the HTTP-server trace logs, because the CGI-bin calls encode the names of the FunStruQL functions and their argument values. Frequently accessed pages can be precomputed by simply adding clauses to the anonymous function; for example, this clause precomputes all new, ISP accounts and can easily be generated automatically:

```
link // Precompute new, ISP accounts.
Precompute() -> "HotList" -> !hotList("ISP", "new")
```

The precomputed pages are cached and immediately available when the user requests them.

Some strategies, however, cannot be inferred automatically. For example, after accessing the appropriate hot list, an account rep scans through the reports for 10-20 of the highest risk accounts, i.e., those with low rank. We could express this strategy by the clause:

```
// Site-generation strategy: precompute
// reports of "new", "ISP" accounts with low rank
where (acct, rank) IN HotList("ISP", "new"),
      (_, _, _, _, _) IN AccountDB(acct),
      rank < 20
link
  Precompute() -> "ReportPage" -> !reportPage(acct)
```

We cannot infer this clause automatically from

server logs, because the logs encode the account numbers of the selected accounts, and on any particular day, a *different* set of accounts has the lowest rank. In this case, the strategy might have to be specified by the site developer. Note that FunStruQL's declarativeness makes it easy for a site optimizer or a developer to specify strategies.

Functions help modularize a query, but they also can introduce redundant computations. FunStruQL's semantics, however, makes it possible to identify and eliminate these computations automatically. For example, the `hotList` function (Fig. 6, line 9) checks that every account in the hot list exists in the account database and then calls `acctInfo`, which queries the same source. When called from `hotList`, `acctInfo`'s query is redundant, but it is not redundant when called from other functions. We can prove that when called from `hotList`, the result of `acctInfo`'s *where* clause is *contained* in `hotList`'s result, and we could optimize `hotList` by inlining `acctInfo` and eliminating the extra query to `AccountDB`:

```
// Optimization: avoid recomputation of acctInfo
fun hotList(type, age) {
  where (acct, rank) IN HotList(type, age),
        (name, street, city, state, zip)
        IN AccountDB(acct)
  link Result() -> "Entry" -> AcctEntry(acct),
    AcctEntry(acct) ->
    { "ReportPage" -> ?reportPage(acct),
      "AcctPage"   -> ?acctPage(acct),
      "Info"       -> AcctInfo(acct),
      "Rank"       -> rank },
    AcctInfo(acct) ->
    { "Acct" acct, "Name" name,
      "Street" street, "City" city,
      "State" state, "Zip" zip } }
```

As with any program optimization, an important problem is deciding *where* to apply optimizations. Although we do not address this problem here, we note that FunStruQL's declarative semantics simplify implementation of query optimizations.

5 Template Language

One premise of STRUDEL's design is that a site's HTML rendering is separable from the site's content and structure. STRUDEL's template language allows the user to specify a site's HTML rendering. A template is a function that maps an object in a site graph to an HTML value. The template's expressions produce HTML values, which are concatenated to produce its result, and are de-

fined by the EBNF rules in Fig. 7. Plain HTML text, the format expression (`sfmt`), conditional expression (`sif`), enumeration expression (`sfor`), and form expression (`sform`) are sufficient for emitting a site graph in HTML. An attribute expression, e.g., `Info.Acct`, denotes the set of objects reachable by edges labeled with the given attributes. An attribute expression implicitly refers to the template's object argument, named `this`, but can refer explicitly to any object variable, e.g., `@this.Info.Acct`. Sometimes more general computation is necessary during HTML generation; the `sjava` construct provides an "escape" into Java, which permits the evaluation of arbitrary Java code.

For each object in a site graph, STRUDEL's generator applies the appropriate template to the object to produce its HTML value. Each object in a site graph has a user-specified *generation mode*: *page* or *page component*; all leaf objects, i.e., atomic values, are page components. Figure 8 contains fragments of the templates for the `Root`, `acctInfo`, and `acctPage` objects in the HTN site. The `Root` and all `acctPage` and `reportPage` objects are realized as pages, and all `acctInfo` objects as page components.

The format expression maps an object to an HTML value. In the `acctInfo` template (Fig. 8), `<sfmt Name>`, refers to the atomic value reachable by the attribute expression `@this.Name`, and is replaced by its HTML value, a string. Format expressions are concise, because the generator uses type-specific rules to determine an object's HTML value. For most atomic values, the object's HTML value is converted to a string. For some atomic values, e.g., those with type `PostScript`, the generator produces a link to its value.

An internal object's generation mode determines how it is formatted. In `acctPage`'s template, `<sfmt Info>`, always refers to an `acctInfo` object `a`, which is a page component, so it is replaced by `a`'s HTML value, but the expression `<sfmt ReportPage link="All Reports">` refers to a `reportPage` object, which is a page, so it is replaced by a link to the appropriate page; the `link` directive specifies the link's tag text.

Some internal objects are *closures*, which represent lazily evaluated functions. In `acctInfo`'s template, `<sfmt AcctPage link=Acct>`, refers to a closure for the lazily evaluated function `acctPage` (as defined in the query in Fig. 6). Its result is a page object, so the generator emits a link that contains a

```

Template : - { Body }
Body      : - PlainHTMLText
           | <sfmt AttrExpr[link = AttrExpr[String]>
           | <sif CondExpr>Body</sif>
           | <sfor ID in AttrExpr[order = (ascend|descend) key = AttrExpr]>Body</sfor>
           | <sform AttrExpr>Body</sform>
           | <sjava>JavaCode</sjava>
AttrExpr : - [@Var.]Attribute{.Attribute}

```

Figure 7: EBNF Rules for the HTML templates.

Root template:

```

<sform HotListForm>
  Choose account age and type:
  Age: <sinput type="text" name="age" value="old">
  Account type: <sinput type="select" name="type">
  <select>
    <sfor t in AcctType>
      <option value="<sfmt @t>"><sfmt @t>
    </sfor>
  </select>
</sform>

```

acctInfo template:

```

<h1>Account #<sfmt AcctPage link=Acct></h1>
<sfmt Name>: <sfmt Street>, <sfmt City>
<sif PostalCode><sfmt PostalCode>
<selse><sfmt Zip>
</sif>

```

acctPage template:

```

<html><sfmt Info><hr>
<sfmt ReportPage link="All Reports">
<table><tr><td>Date</td><td>Total</td></tr>
<sfor e in UsageData.Entry
  order=descend key=Date>
  <tr><td><sfmt @e.Date></td>
    <td><sfmt @e.Total></td></tr>
</sfor>
</table>
</html>

```

Figure 8: Templates for Root, acctInfo, and acctPage objects of HTN site

STRUDEL-specific, CGI-bin expression that encodes the closure function's name and its argument values. When the link is selected at runtime, STRUDEL evaluates the appropriate function and produces the result page. If a closure object produces a page component, the generator applies the closure to produce the object then applies its template to produce its HTML. Note that template expressions are independent of the site-generation strategy, which means the template writer does not have to know how an object is produced. The generator emits the appropriate HTML code whether an object is the result of an eagerly or a lazily evaluated function.

The `sif` expression evaluates a conditional expression and then evaluates the appropriate branch. For example, in the `acctInfo` template:

```

<sif PostalCode> <sfmt PostalCode>
<selse> <sfmt Zip> </sif>

```

tests for a `PostalCode` attribute and emits its value if it exists, otherwise it emits the `Zip` attribute's value.

Objects can have multiple instances of the same attribute, e.g., `acctPage` objects have multiple `UsageData.Entry` attributes. The `sfor` expression binds an object variable to each object denoted by its attribute expression and evaluates its body for each binding. In the `acctPage` template,

```

<sfor e in UsageData.Entry order=descend key=Date>
  <sfmt @e.Date> <sfmt @e.Total>
</sfor>

```

binds `e` to each value of the attribute expression `UsageData.Entry` and emits `e`'s `Date` and `Total` values. The `order` directive sorts objects in either lexicographically increasing or decreasing order; if the objects are internal, the optional `key`

value specifies the attribute that should be used as the sort key. The `sfor` expression above orders the `UsageData.Entry` objects in descending order by their `Date` attribute.

The attribute expression of a `sform` must refer to a *form* object. A form object has free variables, and, like a closure, a target function and some bound variables. In the `Root` template, for example, the `<sform HotListForm>` expression refers to the `HotListForm` object. All of a form's free variables must be bound by `sinput` expressions within the body of the `sform`. In this example, `HotListForm`'s variables `age` and `type` are bound; the possible options for `type`'s value are enumerated by the `sfor` expression. As with closures, the template writer need not know how the target function is evaluated. The generator emits the appropriate "action" value for the HTML form, which includes the target function and its bound variables. Currently, the check that a form's free variables are bound is done dynamically during HTML generation.

Although HTML is the standard output language for a site graph, it is not the only one. STRUDEL can emit a site graph in XML and fewer than 100 lines of STRUDEL's generator are HTML-specific, so other markup languages could be supported with minimal changes to the generator.

6 Evaluation and Discussion

As described in Sec. 1.1, the original HTN site was completely re-implemented using STRUDEL and Daytona, a relational database management system. We compare the total number of files and total number of non-empty, non-comment lines of code for each implementation. Reducing the total line count is not a definitive measure of improvement, but it does indicate the relative effort required for each implementation. Table 1 compares the two implementations. Each source-code file was categorized as primarily site-definition code, HTML-template code, or general-purpose Java code. In the STRUDEL implementation, 66% of the code is devoted to page presentation, but less than 30% is required to define the site. This is encouraging, because the site-definition query contains the potentially reusable part of the specification and is the first and only component that a user would read to understand the site's definition.

In the original implementation, 75% of the code is devoted to site definition, but more importantly, the

Type of code	Implementations			
	STRUDEL		Original	
	# lines	# files	# lines	# files
Site definition	291	1	1198	23
Templates	673	11	42	1
Java code	41		392	1
Total	1005	12	1632	26

Table 1: Comparison of HTN site implementations

code to access data, to define site structure, and to emit HTML code is interleaved, making it difficult to modify or extend. Overall, the STRUDEL implementation is 1.6 times smaller than the original implementation, but if we compare only the code for site-definition, it is more than 4 times smaller. Also, the STRUDEL definition is encapsulated in one file, whereas the original definition was distributed over 23 files.

Unlike the original implementation, the STRUDEL implementation supports flexible site-generation strategies. For example, we implemented by hand some simple strategies, similar to those in Sec. 4.2: precompute frequently accessed hot lists and report pages. These added less than 10 lines to the anonymous function, and in the best cases, reduced page-generation time from 12 seconds to less than 2 seconds. The strategies *extend* the original query with hand-coded optimization rules. Our next challenge is to generate these strategies automatically. HTTP-server trace logs and STRUDEL profiling statistics can provide useful optimization information.

Our general design strategy was to focus on the hardest problems of creating data-intensive Web sites: accessing and integrating data and building the site's content and structure. Our first insight was that these problems are best solved by a declarative query language. Our second insight was that unlike ad-hoc queries in traditional query languages, a StruQL query is also a software artifact, which must be extensible and reusable. We extended StruQL with functions in a way that preserved the simple semantics of StruQL, but that better enabled STRUDEL to support dynamic sites and flexible site-generation strategies.

Overall, FunStruQL's simple, declarative semantics make the language easy to understand and more importantly, easy to analyze. We have already mentioned some unexpected benefits, e.g., declarative specification of error clauses. An important prob-

lem we have not addressed yet is extending FunStruQL with an *update semantics*, i.e., a formalism for specifying updates to a query's domain, and a syntax for specifying updates. Given an update semantics, STRUDEL could support *incremental update* of a site, i.e., identify those parts of the site graph effected by an update and recompute automatically the pages effected.

One common criticism of FunStruQL in particular, and other domain-specific languages in general, is they perpetuate the "Tower of Babel", requiring the user to learn a new language when well-known programming languages can solve the problem at hand. Our response is that FunStruQL's long-term benefits should outweigh the short-term cost of learning the language. Site definitions in FunStruQL are self-documenting and shorter than the equivalent scripting code, making it easier to modify and reuse them immediately. The HTN site substantiates many of FunStruQL's benefits, but we expect that applying it to other sites will reveal other opportunities for improvement.

Remarks. STRUDEL is available from <http://www.research.att.com/sw/tools/strudel>. We thank Sandra Sudarsky for her contributions to STRUDEL's implementation.

References

- [1] S. Abiteboul. Querying semi-structured data. In *Proc. of the Int. Conf. on Database Theory (ICDT)*, Delphi, Greece, 1997.
- [2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley, 1995.
- [3] V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. L. Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, and L. Wood. Document object model level 1.0 specification. Technical Report REC-DOM-Level-1-19981001, World Wide Web Consortium, Oct. 1998.
- [4] K. Arnold and J. Gosling. *The Java Programming Language, Second Edition*. Addison-Wesley, Dec. 1997.
- [5] D. Atkins, T. Ball, M. Benedikt, G. Bruns, K. Cox, P. Mataga, and K. Rehor. Experience with a domain specific language for form-based services. In *Proceedings of Conference on Domain-Specific Languages*, pages 37-49, 1998.
- [6] P. Atzeni, G. Mecca, and P. Merialdo. Design and maintenance of data-intensive web sites. In *Proc. of the Conf. on Extending Database Technology (EDBT)*, pages 436-450, Valencia, Spain, 1998.
- [7] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible markup language (xml) 1.0. Technical Report REC-xml-19980210, World Wide Web Consortium, February 1998.
- [8] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *proceedings of IPSJ*, Tokyo, Japan, Oct. 1994.
- [9] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. Catching the boat with Strudel: experiences with a web-site management system. In *SIGMOD*, Seattle, Wash., June 1998.
- [10] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. Verifying integrity constraints on web sites. In *IJCAI*, 1999.
- [11] D. Florescu, A. Levy, and A. Mendelzon. Database techniques for the world-wide web: A survey. *SIGMOD Record*, 27(3), Sept. 1998.
- [12] P. Fraternali. Tools and approaches for developing data-intensive web applications: a survey. *ACM Computing Surveys*, Sept. 1999.
- [13] R. Greer. Daytona. *Proceedings of the SIGMOD International Conference on Management of Data*, June 1999.
- [14] X. W. Group. Extensible stylesheet language (xsl). Technical Report WD-xsl-19981216, World Wide Web Consortium, Dec. 1998.
- [15] J. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer*, 31(3):23-30, March 1998.
- [16] P. Paolini and P. Fraternali. A conceptual model and a tool environment for developing more scalable, dynamic, and customizable web applications. In *Proc. of the Conf. on Extending Database Technology (EDBT)*, 1998.
- [17] D. Schwabe and G. Rossi. An object oriented approach to web-based application design. *Theory and Practice of Object Systems, Special Issue on the Internet*, 4(4):207-225, 1998.

A Collaboration Specification Language

Du Li and Richard R. Muntz

*Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90024 USA
{lidu, muntz}@cs.ucla.edu*

ABSTRACT

COCA (Collaborative Objects Coordination Architecture) was proposed as a novel means to model and support collaborations over the Internet. Our approach separates coordination policies from user interfaces and the policies are specified in a logic-based language. Over the past year, both the collaboration model and the specification language have been substantially refined and evaluated through our experience in building real-life collaboration systems. This paper presents the design of the specification language and illustrates the main ideas with a few simple design examples. Semantics, implementation, runtime support, and applications are also covered but not as the focus of this paper.

1 INTRODUCTION

Over the last decade, many CSCW (Computer-Supported Cooperative Work) systems have been built. However, in traditional systems, the coordination policies, such as those for access control and concurrency control, are often hard-coded into the system together with the user interfaces. Coordination policies are usually sensitive to the work style and organizational structure of individual groups. Not only do different groups have different needs, but the same group may require different policies in different phases of their collaboration. Therefore, it is important for system support to be flexible and adaptable. But in general, traditional CSCW systems are complex to build and do not easily accommodate changes.

We proposed a novel approach called COCA [16] to model and support collaborations over the Internet. Our approach is different from traditional CSCW systems in the following ways. First, we separate coordination policies from user interfaces in building collaboration systems. Second, we provide an executable specification language to describe coordination policies. Those policies are enforced at runtime by the COCA virtual machine (*cocavm*), a copy of which runs at each participant

site. Third, at runtime participants in the same collaboration take on roles (e.g. the professor and student roles in distance learning). Different roles are each controlled by a different set of rules. As a result, the design and implementation of collaboration tools are greatly simplified. Tools are both easy to build from scratch [17] and to adapt from legacy tools [18]. The same set of tools can be reused in many different scenarios with different policies and without changes to the tools. Moreover, the coordination policies are explicitly and collectively specified instead of being scattered in all the involved components of the system. Systems thus built are more tractable to manage and evolve than those developed in traditional ways.

The notions of role and collaboration are well-accepted in object-oriented systems design and analysis, e.g. [34], [33], and [13]. However, their uses of those two concepts are different from ours, as was concisely illustrated by the following excerpt [27]:

Role models provide excellent separation of concerns due to their focus on one particular collaboration purpose, while traditional class diagrams necessarily interwine all different object collaboration aspects. When composing role models, several aspects of object collaborations can be specified without prematurely committing to a class structure that might turn out to be too rigid later on.

In the CSCW literature, Intermezzo [8], QUILT [14], etc. used the concept of *role*. In Intermezzo and many other systems, roles are used only for access control. We use *role* as a unit to specify a wide range of coordination policies including access control, concurrency control¹, session control, etc. As a result, the language in Intermezzo is much simpler and limited in expressive power, as compared with the language in COCA. QUILT built

¹It is not unusual that preferences are given to some class of participants over another when a conflict arises.

three roles into the group editor: reader, commentator, and co-author, each with different privileges. In COCA, the users are free to specify as many roles as the application requires.

An important difference between groupware and traditional software is the critical need for an efficient and scalable group communication model. Early collaborative systems were generally built based on grouping of multiple point-to-point communications. This approach incurs tremendous overhead on both the message senders and the communication paths, and latency increases with the size of the receiver population. Deering[6] proposed IP multicast as a better solution to this problem. It has been well-accepted in the networking literature (e.g.[10, 23]) that IP multicast is an efficient model for group communication, both for delivery of time-critical media streams and for non-realtime messages. We adopt IP multicast as the group communication model of COCA.

Major novelties of our specification language include the following. It uses roles as a unit to specify a wide range of policies including access control, concurrency control, session control, etc. It provides logic-based language constructs for efficient and flexible group communication. The simple syntax leads to concise and easily understood specifications. It is potentially possible to develop tools to mechanically verify and validate the specifications, to derive test cases, to detect deadlock situations, to reason about role behavior, etc.

Over the past one year, considerable experience with COCA has been accumulated and the system extended to improve the usefulness of COCA in practice. The runtime support system has been running since February 1998. Since the summer of 1998, we have been exploring application domains and building systems. Specifically, COCA has been proved applicable in the following domains: electronic meeting systems[17], online auction systems[18], and so forth. Today over 40,000 lines of code are running, including the core language runtime and applications. The performance has been good despite the fact that the prototype is done in pure JAVA.

The remainder of this paper is organized as follows. We first briefly overview the collaboration model of COCA in the next section. In section 3, we try to give the readers an initial feel of the language with a simple example. Section 4 describes our specification language. A more complete design example is discussed in section 5. Some important implementation issues are discussed in sec-

tion 6. A more detailed comparison with related work is left to section 7. We conclude this paper in section 8 with some directions for future research and development.

2 GENERAL MODEL

A collaboration is a group activity (e.g. a course, a meeting, a game, etc.) which involves a group of participants. A participant is usually a human being but can also be a set of software agents. In a collaboration there are usually multiple participants each playing one or more roles. The concept of **role** refers to the set of participants governed by the same set of coordination policies. In a collaboration, one participant may be playing multiple roles and one role could be enacted by multiple participants at the same time. For example, in a project meeting, one participant plays the project *manager* role and the others are in a normal *member* role. The *manager* is able to see and modify the drawings of the normal *members*, while the latter can see but are not allowed to modify the drawings of other participants.

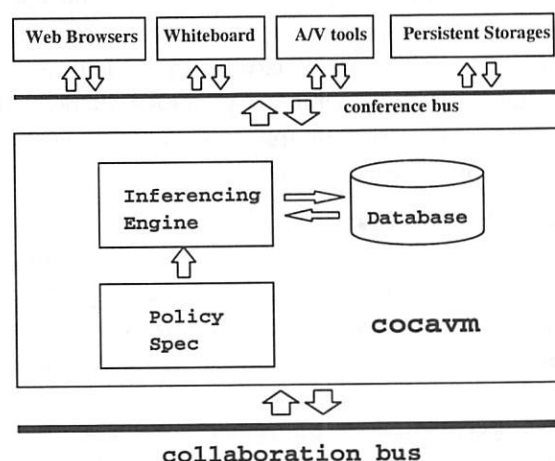


Figure 1: Internal structure of the *cocavm*

In our model, a *cocavm* runs at each participant site to enforce the coordination policies by controlling the interactions between this participant and other collaborators. As shown in Figure 1, a *cocavm* consists of an inferencing engine and an internal database. The inferencing engine monitors messages communicated in the collaboration, firing the active rules unified with the message, and performing actions according to the policy specification. The internal database maintains state information regarding the ongoing collaboration.

Participants in a collaboration use collaboration tools, such as web browsers[18], whiteboard tool[17], floor control tool[16], and the audio/video tools, to collaborate with each other. At each site, the collaboration tools

and a *cocavm* are connected by a conference bus. All the *cocavms* in the same collaboration are connected by a collaboration bus. Those buses contain channels which communicate messages between connected entities. A channel can be both unicast(one to one) and multicast(one to many).

csdr, the collaborative session directory tool, is the simple runtime user interface of COCA. The user can use it to create sessions out of a given collaboration specification, browse existing sessions, and join a session by taking roles from it. In particular, the session creator must provide, among other information, actual IP addresses and port numbers for the channels declared in the collaboration bus. Individual participants must provide this information for the conference bus channels. Such bindings were termed relocation of role and collaboration respectively[16].

3 A SIMPLE EXAMPLE

Floor control is often used in CSCW systems to control the mutual exclusive access to shared resources. There are many different floor control policies[7]. In Figure 2 we specify a centralized policy. There are three roles. The floor *moderator* role (line 8-34) controls who can obtain the floor at any time. A floor *aspirant* (line 36-60) is a participant who needs to apply for the floor from the *moderator*. And the floor *holder* (line 62-63) is the participant who currently holds the floor. At one time only one participant can be the *moderator* and only one can be the floor *holder*. There are no constraints on how many participants can take the *aspirant* role concurrently.

As is shown in Figure 3, participants in the floor *moderator* and *aspirant* roles use their corresponding tools (or graphical user interface, GUI) to interact with each other through the *cocavms*. The tools are connected to the *cocavms* by channels named *local-in* and *local-out* respectively. And all the *cocavms* are connected by a channel *remote*. The *moderator* tool expects two commands from the *cocavm*: one to *request* the floor, and the other to notify the user to whom the floor is *granted* when a coordination decision is made. It sends three commands to *cocavm*: one to *grant* the floor to a given participant, one to *deny* a floor request, and the other to *revoke* the floor from the current floor *holder*. The *aspirant* tool sends two commands to the *cocavm*: one to *request* the floor, and the other *releases* the floor. Messages from *cocavm* to the *aspirant* tool are *grant*, to notify the *aspirant* to whom the floor is granted, and *deny*, to report that the floor request is denied.

```

1. collaboration sfc
2. {
3.     collaboration-bus
4.     {
5.         channel(remote).
6.     }
7.
8.     role moderator
9.     {
10.        conference-bus
11.        {
12.            channel(local-in).
13.            channel(local-out).
14.        }
15.
16.        on-arrival(gate(remote),request(Floor,Agent)) :-
17.            local-out !request(Floor, Agent).
18.
19.        on-arrival(gate(remote),release(Floor,Agent)) :-
20.            local-out !grant(Floor, self),
21.            take holder.
22.
23.        on-arrival(gate(local-in),grant(Floor,Agent)) :-
24.            remote !grant(Floor, Agent),
25.            isa(self, holder),
26.            drop holder.
27.
28.        on-arrival(gate(local-in),deny(Floor,Agent)) :-
29.            remote !deny(Floor, Agent).
30.
31.        on-arrival(gate(local-in),revoke(Floor,Agent)) :-
32.            remote !revoke(Floor, Agent),
33.            take holder.
34.    }
35.
36.    role aspirant
37.    {
38.        conference-bus
39.        {
40.            channel(local-in).
41.            channel(local-out).
42.        }
43.
44.        on-arrival(gate(local-in), request(Floor)) :-
45.            remote !request(Floor, self).
46.
47.        on-arrival(gate(local-in), release(Floor)) :-
48.            remote !release(Floor, self),
49.            drop holder.
50.
51.        on-arrival(gate(remote), grant(Floor, self)) :-
52.            local-out !grant(Floor),
53.            take holder.
54.
55.        on-arrival(gate(remote), revoke(Floor, self)) :-
56.            drop holder.
57.
58.        on-arrival(gate(remote), deny(Floor, self)) :-
59.            local-out !deny(Floor).
60.    }
61.
62.    role holder
63.    {}
64.}

```

Figure 2: A simple floor control policy.

In Figure 2, when an *aspirant* tries to request the floor, the command arrives at *aspirant cocavm's local-in* gate (line 44). It is forwarded to the *moderator* via the "remote" channel (line 45). When this message arrives at the *remote* gate of the *moderator* (line 16), it is forwarded to the *moderator* tool (line 17), leaving the decision to the user.

When the participant in the *moderator* role decides to grant the floor to some participant (line 23), the message is multicast via the *remote* channel (line 24), meanwhile the *moderator* drops the *holder* role if she is currently in that role (line 25-26). When the participant *cocavm* receives the message which grants her the floor (line 51), the message is reported to the tool (line 52) and the floor

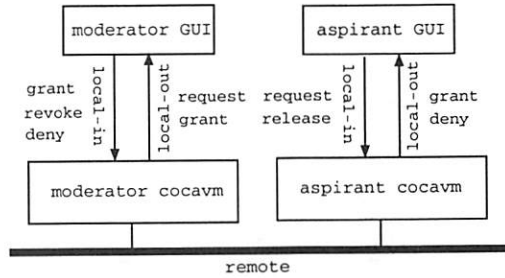


Figure 3: The floor moderator and aspirant roles.

holder role is taken (line 53).

To be concise, when a floor request is denied, the aspirant is notified (line 28-29, 58-59). When the current floor *holder* releases the floor, she drops the *holder* role and meanwhile multicasts the message so that the *moderator* assumes it (line 47-49, 19-21). And when the *moderator* decides to *revoke* the floor, the *moderator* assumes the *holder* role and the current floor holder drops it (line 31-33, 55-56).

The floor *holder* role is fluid. Anyone takes on this role when granted the floor by the *moderator* and drops it as a result of releasing the floor or when the floor is preemptively revoked. At this moment, there is no rule specified for the floor *holder* role. More in-depth discussion of this role resumes in section 5.

4 THE SPECIFICATION LANGUAGE

4.1 The Core Language

Following the conventions of Prolog, variables begin in upper case. Constants, functors and predicates begin in lower case. Anonymous variables are denoted by underscores.

Definition 4.1 A term is defined inductively as follows:

1. a variable is a term
2. a constant is a term
3. if f is an n -ary function symbol and T_1, T_2, \dots, T_n are terms, then $f(T_1, T_2, \dots, T_n)$ is a term (called a compound term), $n \geq 0$. For convenience, we use f/n to denote a functor f with arity n .

Definition 4.2 A formula is defined inductively as follows:

1. if p is an n -ary predicate symbol and T_1, T_2, \dots, T_n are terms, then $p(T_1, T_2, \dots, T_n)$ is a formula (called

- an atomic formula or more simply, an atom), $n \geq 0$. We use p/n to denote a predicate p with arity n .
2. if F is a formula, so is (not F).

Definition 4.3 A literal is an atom or the negation of an atom. A positive literal is an atom. A negative literal is the negation of an atom.

Definition 4.4 If q is a positive literal, p_1, p_2, \dots, p_n are literals, $n \geq 0$, then

$$q : -p_1, p_2, \dots, p_n.$$

is a rule. Atom q is called the head, and p_1, p_2, \dots, p_n combined is called the body of this rule.

4.2 Database Operations

The above-defined core language is no different from Prolog. To avoid unauthorized modification to coordination policies and for more efficiency, however, COCA deliberately separates the rule base and the database. We use a set of database operators rather than **assert** and **retract** in Prolog. Those Prolog predicates do not distinguish predicate definitions and database facts.

Definition 4.5 We define database formulas as follows. If T is a compound term, the following atomic formulas are database formulas. Keywords **query**, **add**, and **delete** are database operators.

1. **query** T : evaluates to true if compound term T is unified with any compound term in the database;
2. **add** T : add compound term T to the database, evaluates to true if it succeeds;
3. **delete** T : delete a compound term unified with T from the database nondeterministically, evaluates to true if it succeeds.

Those operators are atomic, synchronized, and nonblocking. In our implementation, database facts are hashed by functor names. Mutual exclusion is enforced so that no two operators can work at the same time on compound terms with the same functor name.

We can extend this set to support blocking operations as well. For example, **query^{sync}** and **delete^{sync}** can be used to denote blocking database operations respectively. Although they are only for synchronization within the same *cocavm*, those operators have a close relationship to the tuple space operations in Linda[12]. **query^{sync}** corresponds to **rd**, **delete^{sync}** to **in**, **query** to **rdp**, **delete** to **inp**, and **add** to **out**.

The role of the database in COCA is two-fold. First, it provides an ephemeral memory (as opposed to persistent storage systems such as a relational database) for capturing and recording the state information regarding the ongoing collaboration, based on which many a coordination decision is made. Second, it implements a synchronization mechanism based on which concurrent activities in each *cocavm* are coordinated.

4.3 *cocavm* Identification

A *cocavm* has a unique identification. This id is denoted by keyword **self**. It could include the following information: host name, user login name, command port number on which this *cocavm* receives commands, a time stamp when this *cocavm* is launched, a list of roles this *cocavm* is enacting, etc.

4.4 Communication

4.4.1 Channel and Gate

The following channel declaration defines a channel with name *C*:

channel(*C*).

We use the term **gate** to denote a service access point to a communication channel. Gates use the same name as the corresponding channels.

Definition 4.6 *We define the following communication formulas as complementary to Definition 4.2, where operators “!” and “?” are called offer and accept respectively.*

1. $G ! (T_1, T_2, \dots, T_n)$
2. $G ? (T_1, T_2, \dots, T_n)$

(1) sends out a list of terms through gate *G*, (2) blocks until a list of terms arriving at gate *G* unify with T_1, T_2, \dots , and T_n as a whole.

Definition 4.7 *We define active rules as having the following form:*

on-arrival(**gate**(*G*), T_1, T_2, \dots, T_n) :-
 P_1, P_2, \dots, P_m .

4.4.2 Dynamic Grouping

A role name say *R* can be used as a channel without being explicitly declared in the collaboration bus to deliver messages to all the participants *currently* in the role. However, since only participants enacting role *R* can receive those messages, we require that only *R* can define **on-arrival** rules at the gate it represents.

Predicate **channel**/2 can be used to declare channels dynamically. The following names a channel *C* which connects a group of participants denoted by $Agent_1, Agent_2, \dots, Agent_n$:

channel(*C*, [$Agent_1, Agent_2, \dots, Agent_n$]).

$Agent_i$ can be a defined static channel name, a role name, some **self**, or even another defined dynamic channel name. A channel (static or dynamic, excluding that denoted by a role name) can be later redefined. The same channel predicate, if the second argument is a variable, can be used to get the list of agents connected by a given channel.

An offer predicate can send messages through a variable channel like the following.

$C ! (T_1, T_2, \dots, T_n)$

However, the variable *C* must have been bound to an agent (some **self**), an agent group, a role name, or a declared channel name.

There are two ways to receive messages sent via a dynamical channel. One is to define active rules at the gate denoted **self**. Another way is to define an active rule in which the gate name is a variable. When a message arrives not unifying with any active rules with constant gate names, the variable gates rules will be tried.

4.5 Events

Events include message arrivals, timer signals and user posted events. Messaging events were discussed in the previous subsection. Here we discuss the latter two.

4.5.1 Event Generator

The following predicate sets a timer.

set-alarm(*TimerName*(*FireTime*)).

After a timer is set up, it fires only once and posts an event with the same name as the timer. If it is to be fired repeatedly, it must be reset each time. A timer event is always asynchronous. The firing time of a timer can be a relative time, i.e, some period from now, or an absolute time set on a particular time or date.

The following predicate posts a user-defined event either synchronously or asynchronously.

post-event^{*sync|async*}(*EvtName*(T_1, \dots, T_n)).

4.5.2 Event Handler

When an event occurs, the corresponding event handler or active rule is fired. If the event is asynchronous, the event handler will be executed in parallel with the current thread. If it is synchronous, the firing will be equivalent to calling a predicate in the current thread. An

event handler is defined as below:

```
on-event(EvtName( $T_1, \dots, T_n$ )) :-  
     $p_1, \dots, p_m$ .
```

4.5.3 wait-for Predicates

Predicate **wait-for/1** can be used by the current thread to wait for some event to occur. In previous releases of COCA, when a message was expected from a certain object, we needed an active rule to wait for it. This new construct can make it simpler and clearer. For example, when a message is sent to an object and an answer is expected from it, we simply specify the following.

```
 $p$  :- ...  
    g !(SomeRequest),  
    wait-for(SomeEvent),  
    ...
```

A second argument can be added to **wait-for** which specifies the timeout period before **wait-for/2** can be failed. Any nonnegative number is valid or the symbol " ∞ " can be used to denote infinity. If it is an unbound variable, when the predicate succeeds this variable is bound to the length of time from when the **wait-for** was executed until the event occurred.

4.6 Role

4.6.1 Definition

At least one role must be defined in a collaboration. Each role definition has the following form.

```
role <role name>  
{  
    [ conference-bus { <channel declarations> } ]  
    <rules>  
}
```

A conference bus declaration is optional. We can define roles which do not interact with other local components. The conference bus is used for communication between the *cocavm* and the collaboration tools for each role. So channel names declared in a role are local to that role. Predicate **isa/2** can be used to test if a given participant (some **self**) is currently in some role.

4.6.2 The Daemon Role

A collaboration typically defines a special role called *daemon*, which controls for example who is allowed to take which roles and how many participants can take a certain role². When a participant attempts to take a role

²[27] described four possible role constraints: role-dontcare, role-implied, role-equivalent, and role-prohibited. Actually more possibilities exist. For example, one can not take on the student role in a given course if she was once the teaching assistant of the

from an ongoing session, the daemon of that session is contacted. Possibly authentication is performed. Only when the participant is qualified by the session control policy does she obtains the rule set specified for that role. A session without a daemon role will be open to all, i.e., any participant can take any role.

The session control may not necessarily be centralized. It is possible for a session to have multiple daemons, e.g. for availability and scalability. The user can specify a daemon role which can be taken by multiple agents.

4.6.3 Constructor and Destructor

Operators **take** and **drop** can be used to take a role and to drop a role respectively. In the following, R stands for the name of some role and $n \geq 0$.

1. **take** $R(T_1, T_2, \dots, T_n)$
2. **drop** R

To perform initialization when a role is taken and to clean up when a role is dropped, we define constructor and destructor rules as follows.

1. **on-take**(T_1, \dots, T_n) :- p_1, \dots, p_m .
2. **on-drop** :- p_1, \dots, p_m .

Arguments of the constructor rules are optional. For each role we can define one or more constructor rules which are fired at a *cocavm* when a participant takes on this role. Only the constructor whose arguments unify with those of the **take** predicate is chosen to execute upon initialization.

However, we define at most one destructor for each role. The destructor is fired at a *cocavm* when the role is dropped. At this point, it becomes inappropriate for any rules defined for that role to continue their execution. Our strategy is to mark all the arrival message and event queues in the *cocavm* so that no more messages or events will be processed after the mark. When the *cocavm* comes to a quiescent state, the destructor is fired to clean up.

4.7 Collaboration

A **collaboration** is defined in the following form:

```
collaboration <collaboration name>  
{  
    [ collaboration-bus { <channel declarations> } ]  
    <role definitions>  
}
```

same class. Sometimes it is up to the consensus (voting) of the admitted participants to decide whether to admit a new participant, etc.

An operator “ $::$ ” is used for identifying the scope of a name. The normal form is $S :: R :: C$, where S stands for collaboration or session name, R for role name, and C for channel name or predicate name. The prefixes can be omitted if there is no confusion.

A collaboration may consist of sub-collaborations. For example, in activities such as conferences and classrooms, it is not uncommon that participants with similar interest form subgroups. So the language should provide constructs to facilitate establishing new sessions out of given collaboration types and tearing down existing sessions. The following operators are defined for this purpose.

1. **create** *Session, Collaboration*
2. **destroy** *Session*

After a session is created, the user can join it by command “**take** *Session :: Role*”.

4.8 Policy Composition

To better capture and specify coordination policies, we often need to decompose a complex collaboration into smaller pieces, and compose existing pieces to form a larger one. Here we introduce some constructs for policy composition. These constructs provide a basis for defining a library of reusable coordination policies and predicates.

4.8.1 Parameterization

The first construct is for definition of policy templates. For example, in defining the floor control policy in section 3 to control the drawing floor, we discover that it also applies to the audio floor and the video floor in multimedia collaborations[7]. It would be advantageous then to formalize the floor name so that the same specification can be reused.

```
template<Floor>
collaboration sfc
{
  ...
}
```

Then we can actualize this policy template by replacing the formal “Floor”, such as *sfc*<drawing-floor> and *sfc*<audio-floor>, which becomes the name of a collaboration type. Multiple formal names in a policy template are allowed. The occurrences of each will be substituted by corresponding actual names.

4.8.2 Inheritance

The second construct is for policy inheritance. For example, if we have already defined collaborations A and

B . Now we want to define a new collaboration C . And it turns out we can reuse the definitions in A and B . So C just needs to extend those two existing collaborations.

```
collaboration  $C$  extends  $A, B$ 
{
  ...
}
```

The collaboration bus channels and roles of the resulting collaboration C are a union of those defined in A , B , and itself. Wherever there is a name conflict, the above scoping operator “ $::$ ” and the appropriate prefixes will be applied to resolve it automatically.

Roles can also be **extended** from roles defined in ancestor collaborations or other roles defined in the same collaboration. In the above example, role r_3 in collaboration C can extend both role r_1 in collaboration A and role r_2 in collaboration B as follows.

```
role  $C::r_3$  extends  $A::r_1, B::r_2$ 
{
  ...
}
```

In a sequel, conference bus channels and rules of $C::r_3$ will be a union of those defined in $A::r_1$ and $B::r_2$ respectively.

4.8.3 Polymorphism

The ordering of rules is important. The rule set of a derived role is always put before those of its parents. When a predicate is evaluated, the rule set of the role which is on the lowest level of the inheritance hierarchy is consulted first. If several parents exist, inherited rules are put by the order in which the parents appear. If there are multiple layers of inheritance, the same rule applies recursively. Inheritance of collaborations or roles are acyclic. In the case that a role is inherited more than once in the same layer or different layer, the rule set of that role is included only once.

Constructor and destructor rules are invoked when a role is taken or dropped. Those defined in the lowest hierarchy are considered first. In the above example, in the constructor of $C::r_3$, if the user wants to execute the initialization code of role $A::r_1$ as well, the constructor of the latter should be called explicitly as follows.

```
 $C::r_3::on\text{-}take(T_1, \dots, T_n) :- \dots$ 
 $A::r_1::on\text{-}take(T_1, \dots, T_n),$ 
...
```

In COCA we use a role name as an implicit communication channel and active rules can be defined at the gate

it represents. For example such an active rule can be defined for role $A::r_1$. When a message is sent to gate $A::r_1$, participants in role $C::r_3$ should also receive it. The reason is that the active rule defined at gate $A::r_1$ is inherited by $C::r_3$ and becomes a part of it.

5 A COMPLETE EXAMPLE

Here we first consider a whiteboard meeting, then discuss a concurrency control policy, and show how these two policy modules can be composed into one for project meetings. We assume that all participants join a meeting at about the same time. It is impossible to have one short example to illustrate all the language constructs. Policies to handle late joins and other examples can be found in [15].

5.1 Whiteboard Meeting

We have implemented a whiteboard tool[17] which can be used either in a single-user mode or in a multi-user mode. In the former case, the user can draw, delete, and modify objects such as lines, circles, and free-hand pictures, load images, etc. In a multi-user mode, COCA can be used to enforce specified coordination policies such as those for access control, concurrency control, and session control. For example, a policy can be specified where, when a user attempts to annotate an object, the command is sent to the owner of that object. If the owner personally does not like the annotation, then it will not appear on both parties' whiteboards and will not be propagated to other sites. In the following, however, we only consider a simplest case in which everybody can do anything and what you see is what I see (WYSIWIS). In[17] we showed how the same whiteboard tool can be used under many different coordination policies.

```

collaboration meeting
{
  role drawer
  {
    conference-bus
    {
      channel(local-in).
      channel(local-out).
    }

    on-arrival(gate(local-in), Cmd) :-
      drawer !Cmd.

    on-arrival(gate(drawer), Cmd) :-
      local-out !Cmd.
  }
}

```

Figure 4: A policy for WYSIWIS meeting.

In this specification, each command from the whiteboard is propagated to all participants and any commands from other sites are sent to the local whiteboard, both without examination. However, conflicts can arise. For example several participants may happen to modify the same ob-

ject on the screen. Some concurrency control policy must be enforced to resolve such conflicts. Here for simplicity, we adopt a floor control scheme in which participants take turns to draw on the whiteboard. Specification of other concurrency control policies (e.g. [9]) can be found in [15].

In a floor control scheme, a participant must have the floor to draw on the whiteboard. We change the first rule to the following.

```

on-arrival(gate(local-in), Cmd) :-
  isa(self, holder),
  remote !(Cmd).

```

Each time the local participant attempts to draw an object, the command is not propagated to other sites unless the participant currently is the floor *holder*. Note this policy is not complete. It says nothing about how to become a floor *holder*.

5.2 Floor Control

In section 3, we discussed a rather simple floor control specification. However, the *sfc* policy thus defined may not work well in large-scale collaborations over the Internet. Here we extend it to handle some exceptions which are typical in such situations.

In a floor control policy it is important to make sure there is always one and only one participant in the *moderator* role. If the *moderator* is lost, for example due to process failure or network partition, some other participant must be chosen to become the *moderator*. There is a similar issue with regard to the floor *holder* role. If the floor *holder* is detected lost, then the *moderator* must be able to reproduce a floor so that the collaboration can continue. As specified in Figure 5, we use a soft-state protocol for this purpose. The *moderator* and the floor *holder* periodically send out a liveness report, from which the other participants know their liveness and try to repair if there is a loss.

The floor *holder* adds to its database the fact that it is the current *holder* of the floor. This fact is deleted when the role is dropped. Then a timer is set so that it multicasts a liveness report every 10 seconds.

When the *moderator* role is taken, the participant automatically assumes the floor *holder* role. Here two timers are set. The "reporter" sends out the liveness report every 10 seconds. And the "checker" checks the database records every minute. If no liveness report is received from the floor *holder* in one minute, the *moderator* itself

```

collaboration fc extends sfc
{
  role holder extends sfc::holder
  {
    on-take :-
      add holds(self, floor),
      set-alarm(reporter(10000)).

    on-drop :-
      delete holds(self, floor).

    on-event(reporter(Period)) :-
      remote !live(self, holder),
      set-alarm(reporter(Period)).
  }

  role moderator extends sfc::moderator
  {
    on-take :-
      take holder,
      set-alarm(reporter(10000)),
      set-alarm(checker(60000)).

    on-event(reporter(Period)) :-
      remote !live(self, moderator),
      set-alarm(reporter(Period)).

    on-arrival(gate(remote), live(_, holder)) :-
      time(Now),
      add live(holder, Now).

    on-event(checker(Period)) :-
      query live(holder, LastT),
      time(Now),
      Now - LastT >= Period,
      take holder,
      set-alarm(checker(Period)).
  }

  role aspirant extends sfc::aspirant
  {
    ...
  }
}

```

Figure 5: The extended floor control policy.

assumes the floor *holder* role. When a report comes from the *holder*, however, the database is updated accordingly.

The *aspirant* role can be specified similarly. The differences are that an *aspirant* does not need to send the periodic liveness reports and that it must detect the *moderator* loss. When the *moderator* is discovered to be lost, a new one must be chosen from the participants. A number of policies can be applied here depending on the taste of the participants. For example, the decision can be made by voting, by the alphabetic ordering of participants' names, etc. For reasons of space, the *aspirant* specification is listed in [15] instead.

```

collaboration projmeeting
extends meeting, fc
{
  role manager extends moderator, drawer
  {}

  role member extends aspirant, drawer
  {}
}

```

Figure 6: The synthesized project meeting policy.

5.3 Synthesis

Suppose we want to have a project meeting in which participants take turns to draw on the whiteboard. We

should somehow put together the above specified two collaboration types. In Figure 6, the collaboration we want, *projmeeting*, extends collaborations *meeting* and *fc*. Only two roles are available here. The project *manager* is composed from the floor *moderator* and the *drawer*. And the normal project *member* is a composition of the *aspirant* and the *drawer*. So the *manager* controls the floor. Participants in both roles are free to draw on the whiteboard once they become the floor *holder*. Roles inherited, e.g. *drawer*, *moderator*, *aspirant*, and the floor *holder* are made invisible to participants of the collaborative sessions created out of this collaboration.

6 DISCUSSION

This section discusses some important implementation issues. We first briefly present a well-accepted algorithm in subsection 6.1 for maintaining temporal relationships between messages. The support of transaction is discussed in subsection 6.2. Subsection 6.3 discusses the semantics of the event constructs introduced in section 4. In subsection 6.4 we show how to specify messaging policies for more flexibility. Subsection 6.5 illustrates how to specify some predicates which were introduced as built-in predicates in section 4 using the language itself. Backtracking is discussed in subsections 6.6.

6.1 Logic Clock

Each *cocavm* has a logical clock. When a message is sent out, the local logic clock is advanced by one. When a message is received from another *cocavm* with logic time t_1 , and the logic clock of this *cocavm* reads t_2 . If $t_1 \leq t_2$, then we set $t_2 \leftarrow t_1 + 1$. This scheme maintains the causal relationship of messages across *cocavms* and can be easily extended to support a consistent total ordering of all messages[32].

6.2 Transaction

Within the *cocavm*, there could be multiple threads of execution. It is necessary to have some synchronization mechanism between concurrent threads. For this purpose, operators of the internal database are atomic and synchronized. And we further introduce a special symbol "@" as syntactical sugar to define atomic predicates and sequences. An atomic sequence is analyzed before execution. All the database functors (or relations) involved in such a sequence will be locked when the critical section is entered. The lock is released when the sequence succeeds or is failed. A thread blocks if it can not obtain the lock. To prevent deadlocks, all the involved functors must be locked or none.

An atomic sequence corresponds to the concept of trans-

action. To support transactions in the internal database, we would have to support rollback, i.e., when one predicate in the atomic sequence fails, the effects of the whole transaction must be undone. This is not hard to support though. For example, we can analyze the sequence and make a backup copy of those functors that could be affected. Once an operation fails, we can overwrite the original copy with the backup copy. If the whole transaction succeeds, we simply throw away the backup copy.

6.3 Event Semantics

The general principle is to finish the execution of a rule as fast as possible. When an event is posted (synchronously or asynchronously), the blocking **wait-for** predicates are given higher priority than the **on-event** rules. If there are several threads waiting for an event, then only one of them is picked up nondeterministically to evaluate its **wait-for** predicate. If no such thread is eligible to continue, then the **on-event** rules are considered.

Messages are processed in the same way. Predicate **accept** corresponds to **wait-for** and **on-arrival** to **on-event**.

6.4 Messaging Policy

In our current implementation, a thread is waiting for message arrivals at each gate. By default, messages are processed in an FIFO order. It is sometimes useful to change this order for example according to the role of the sender, the content or the logic time of the message, etc.

We can easily specify the messaging policies. The **on-arrival** rules defined at each gate can insert the arrived messages to a central queue maintained in the database. An (infinite) loop there can activate various event handlers which process messages in the queue. The following is only one way to implement a messaging policy, where implementation of the underlined predicates are left to the user. These two predicates are policy-dependent.

```
on-take :-
    add queue([]),
    post-eventasync(execution-loop).
on-arrival(gate(G), ...) :-
    @enqueue(G, args(...)).
enqueue(G, Msg) :-
    delete queue(L0),
    insert(L0, msg(G, Msg), L1),
    add queue(L1).
on-event(execution-loop) :-
    dequeue(G, Msg),
```

```
post-eventsync(msg(G, Msg)),
post-eventsync(execution-loop).
```

Timer events and user-posted events can also be processed through the queue. To do this, the built-in **post-event** predicates must be overridden to insert the events into the queue. But care must be taken so that the **post-event** predicates used above still work as intended.

This scheme has significant advantages. First, the need for concurrency control inside the *cocavm* is ameliorated, since at any moment there is only one rule active in a *cocavm*. We can maintain a central event queue, which includes all message arrivals, timer or user-posted events, instead of a pure message queue. Second, it is easier to stop the execution of the current rule set say when the participant wants to switch to a different role or a different policy (rule set) at runtime[19]. When the command to freeze the runtime state is received, each *cocavm* can simply neglect messages with logical time later than that of the freeze command.

This design assumes that predicates in event handlers can finish within a reasonable time. When a blocking predicate is called, it is still feasible to specify a sophisticated threads scheduling policy to suspend the waiting thread and continue to process another event. But if a predicate takes too long or loops forever, however, all the other events must suffer even indefinitely, as such situations are hard to detect mechanically.

6.5 Some Predicates Revisited

A number of database predicates can be easily defined by atomic sequences and atomic predicates, for example, **update**, **collect**, **deleteAll**, etc. While **collect** and **deleteAll** must be implemented assuming a total ordering of facts, **update** does not. Predicate **update** is implemented simply by an atomic sequence as follows.

```
update(OldT, NewT) :-
    @(
        delete OldT,
        add NewT
    ).
```

The blocking database operators introduced above can be implemented with **wait-for**. For example, predicate "**query**^{sync} f(a, b)" can be defined by the following. Operator **delete**^{sync} can be defined similarly.

```
querysync f(a, X) :-
    query f(a, X).
querysync f(a, X) :-
    wait-for(add-f(a, X)).
```

However, when a fact say “f(a, b)” is added into the database, an event “add-f(a, b)” must be posted either by the user or by the runtime system.

The **accept** operator can also be elegantly defined in terms of **wait-for** as follows, where *g* is the name of a gate.

```
g ? (X, a) :-
    wait-for(g(X, a)).
on-arrival(gate(g), X, a) :-
    post-eventasync(g(X, a)).
```

Predicate **wait-for** also provides a possible means to implement locking and unlocking of database relations, as follows.

```
lock(Object) :-
    query locked(Object),
    wait-for(unlocked(Object)),
    add locked(Object).
lock(Object) :-
    add locked(Object).
unlock(Object) :-
    delete locked(Object),
    post-eventasync(unlocked(Object)).
```

Once an “unlocked(Object)” event is posted, only one thread waiting for it is woken up nondeterministically to consider its **wait-for** predicate. So this definition guarantees that the lock not be grabbed by several threads at the same time.

Since those predicates can be specified, we do not really have to implement them.

6.6 Backtracking

The essence of don’t-know nondeterminism is that failing computations “don’t count” and only successful computations may produce observable results. The don’t-care interpretation of nondeterminism, on the other hand, requires that results of failing computations be observable. Hence a don’t-care nondeterministic computation may produce partial output.

Don’t-care nondeterminism is essential in modeling concurrent interactive systems[31]. In concurrent systems it would be too expensive to backtrack a choice, even if it proves to be wrong. The reason is that a choice, and the actions done afterwards, have already influenced the environment: to maintain consistency the whole system should backtrack, but this is too inefficient. It is rather preferable to provide mechanisms to control the choices, so to avoid as far as possible that wrong decisions are taken[5].

COCA has constructs for communication, asynchronous event posting, and transactions which produce side effects. We take both the don’t-know and don’t-care interpretation of nondeterminism. Suppose the following is the execution trace of an active rule where $m, n \geq 0$ and q_1 is the first predicate with side effect.

$$h \text{ :- } p_1, \dots, p_m, q_1, \dots, q_n.$$

This is similar to the guarded Horn clauses notation in concurrent logic programming languages[31, 5]. The execution of an active rule is implicitly divided into two parts. p_1, \dots, p_m corresponds to the guard part, and q_1, \dots, q_n the body. A failed predicate in the guard part may incur backtrack. But once the body part is entered, the side effects ever generated cannot be undone.

As was introduced in subsection 6.2, failure of a predicate within a transaction causes the whole transaction to rollback. Not to contradict the don’t-care semantics, we restrict that the definition of a transaction should not call (directly or indirectly) predicates that communicate or post events.

7 RELATED WORK

7.1 Other Approaches in Collaboration Specification

Trellis[11] and DCWPL[3] also advocated separating coordination from computation so that the former can be specified in a more declarative way and interpreted at runtime.

DCWPL provides a rather ad hoc scripting language with facilities for modeling and controlling access to shared artifacts. It allows for definition of a fixed set of artifact attributes such as “Authorized”, “MaxInstance”, and trigger-like attributes such as “Preexecution”, “Postexecution”. Even if we can presume that the set of attributes provided is sufficient to capture every possible artifact in all collaborations so that we do not need to extend the language under any circumstances, the problem is that whenever a policy or function is not directly available in the declarative language DCWPL, the user still needs to program it in a procedural language. In this way the declarativeness of the language is undermined. The language we propose in COCA, however, is self-contained. The users can specify whatever coordination policies they want without constantly resorting to a procedural language, assuming the tools have the required functionality. In this way the declarative feature is reserved.

Trellis used a variant of Petri Nets, CTN or colored timed

nets, to specify group interaction protocols. Trellis is a client/server architecture in which a centralized controller processes service requests from clients according to the specification. We argue that a centralized architecture like Trellis may work well for small groups which primarily exchange textual messages. But for large groups with hundreds of participants, especially when multimedia data is widely communicated, it is not clear how it is modeled and handled in Trellis and what level of performance can be expected. It is also not clear, at least in the paper, how CTN models the collaboration of large groups whose participants are highly fluid, and how exceptions, such as floor loss and unexpected loss of the current floor holder, are handled in a Trellis specification.

[26] attempted to use LOTOS[22] to specify a group drawing tool. However, our observation is that LOTOS and other process algebra based formal methods are not convenient for specifying collaborations. In LOTOS, communicating components must be explicitly connected by parallel composition operators. This proved sufficient for some situations where communicating components are fixed and relatively small in number. But in many collaborations, only the participant types (namely the roles) are fixed, while the number of involved participants is large and fluid, the exact number cannot be determined at specification time. LOTOS is not directly convenient for specifying such situations. And even for small and fixed-population collaborative situations, LOTOS is also not always convenient. For example, in order to model situations where a process offers a value and several other processes accept it, LOTOS uses a multi-way synchronization which fails if even one process fails to synchronize at that point. Many collaborations can actually tolerate such exceptions by allowing processes that have received the value to proceed while ignoring or trying to re-transmit the lost data to processes which are temporally unavailable due to communication link delay or failure. We replace the parallel composition operators in LOTOS with the collaboration bus and the conference bus, synchronization among communicating components are achieved by explicit message passing.

7.2 Coordination Languages

Moses[24] was originally intended to make the Linda[12] communication safer by ensuring that the interaction of each process with the shared tuple space be managed by a controller. Each controller enforces a set of rules to capture the following two events: when a message

is sent out by the local agent, and when one arrives at this agent, the local control state is checked, transformation is possibly performed, then this message is either forwarded to the tuple space, or delivered to the local agent, or blocked according to the rule definition. Our work is different from Moses in the following ways: (1) we explicitly divide participants in a policy group by roles. Different roles are controlled by different set of rules rather than all participants governed by the same set of rules as in Moses. Messages can be directed to roles (fluid sets of participants) as well as individual participants. We further allow participants to dynamically join and leave a collaboration by taking and dropping roles, and support the dynamic modification of rules at runtime. (2) We borrowed the concept of "gate" from LOTOS to denote the points where our controller, the COCA virtual machine, interacts with its environment, and define active rules upon the arrival of messages at these gates. In this way, the controller can monitor messages arriving at multiple different gates instead of just one to and one from the tuple space as defined in Moses.

Both the tuple space in Linda and the internal database in COCA are associate memory for coordination between concurrency activities. Linda is often criticized for its overheads to maintain a tuple space. Such overheads become more aggravated when situated in large-scale distributed systems. In COCA, there are actually two levels of concurrency. First, activities of all the participant *cocavms* are concurrent. Their coordination and synchronization are achieved by explicit, asynchronous message passing. Second, within each *cocavm*, multiple threads are executed concurrently. Their coordination is primarily through database operations which are correspondent to those in Linda. The cost of maintaining such an internal database is trivial, if not none.

In Manifold[1], ports are unidirectional and are considered as part of the definition of a component or *manifold*. Channels are established between the output port of a manifold to the input port of another manifold. Although this kind of connections are not restricted to be one-to-one, multi-point communication must be fulfilled by connecting one output port to multiple input ports. So it is not the multicast bus architecture as we are using.

7.3 Logic Programming

In concurrent logic programming (CLP) languages[31], each goal atom is viewed as a process. Concurrent processes communicate and synchronize via instantiation of shared logical variables. In COCA, however, an active

rule is fired upon the arrival of a message at a gate or more generally the occurrence of an event. Concurrency only occurs among those active rules. We do not deliberately pursue the fine-granule parallelism in CLP languages. Concurrent threads in the same *cocvm* synchronize through operations against the shared internal database and different *cocavms* communicate through explicit message passing.

A number of object-oriented extensions to Prolog have been proposed[4]. As was discussed in section 4.8, the extensions we made in COCA, however, are no more than syntactical sugars. They do not have any impact on the language semantics.

Delta-Prolog[25] augmented Prolog with CSP-like communication primitives. But it is not reactive, since it may backtrack on communication[31]. [30] studied the embedding of Linda in a CLP language. Concurrent constraint programming[28] embodies explicit mechanisms for communication and synchronization consisting of two kinds of actions, *ask* and *tell*. Semantics about concurrency and communication in logic has been well-studied in the literature, e.g. [25], [2], and [29].

8 CONCLUSIONS

This language is not completely new. It has language elements from Prolog, concurrent logic programming languages, process algebras, Linda, and object-oriented programming languages. It also has concepts such as atomic sequence (transaction) and events. To better model collaborations it supports the concept of role and role operators. It would be interesting and challenging to develop a formal semantics of this language so that the many pieces from various sources can be reconciled in one elegant mathematic framework.

The latest version of COCA includes the following extensions. A set of role operators were introduced in [19] to model runtime dynamics in collaborative systems, e.g., operators for switching between different roles, transferring a role between participants, evolution of coordination policies on the fly, in addition to the take role and drop role operators discussed in Section 4.6. Coordination policies specified in COCA can be verified, as is discussed in [20]. A new application was explored in [21] to exercise the "dynamic grouping" feature discussed in Section 4.4.2. Further information on COCA and its applications can be found in [15].

ACKNOWLEDGEMENT

Valuable comments from Dr. Lalita J. Jagadeesan at the Bell Labs and Dr. Stott Parker at UCLA improved the

presentation quality of this paper.

REFERENCES

1. F. Arbab, I. Herman and P. Spilling, An overview of Manifold and its implementation. *Concurrency: Practice and Experience*, 5(1):23-70, February 93
2. Anthony J. Bonner and Michael Kifer, Concurrency and Communication in Transaction Logic, In *Proceedings of the Joint International Conference and Symposium on Logic Programming (JICSLP)*, Bonn, Germany, Sept. 1996
3. Mauricio Cortes and Prateek Mishra, DCWPL: A Programming Language for Describing Collaborative Work, *ACM CSCW'96 Proceedings*.
4. Andrew Davison, A Survey of Logic Programming-based Object-Oriented Languages, Technical Report 92/3, Department of Computer Science, University of Melbourne, Australia
5. Frank S. de Boer and C. Palamidessi. From Concurrent Logic Programming to Concurrent Constraint Programming. In G. Levi (editor), *Advances in logic programming theory*. Oxford University Press, 1993.
6. Steven Deering, Multicast Routing in Datagram Internetworks and Extended LANs. *ACM Transactions on Computer Systems*, 8(2), May 1990
7. H.-P. Dommel and J.J. Garcia-Luna-Aceves. Floor Control for Multimedia Conferencing and Collaboration. *ACM Multimedia'97*, 5(1), Jan. 1997
8. W. Keith Edwards, Policies and Roles in Collaborative Applications, *ACM CSCW'96 Proceedings*
9. C.A. Ellis and S.J. Gibbs. Concurrency Control in Groupware Systems. *ACM SIGMOD'89 Proceedings*, Portland, Oregon
10. S. Floyd, V. Jacobson, C. Liu, S. McCanne and L. Zhang, A Reliable Multicast Framework for Lightweight Sessions and Application-Level Framing, *ACM SIGCOMM'95 Proceedings*, Boston, 1995
11. Richard Furuta and David Stotts, Interpreted Collaboration Protocols and Their Use in Groupware Prototyping, *ACM CSCW'94 Proceedings*.
12. David Gelernter, Generative Communication in Linda, *ACM Transactions on Programming Languages and Systems*, Vol.7, No.1, Jan. 1985

13. G. Gottlob, M. Schrefl, and B. Röck. Extending Object-Oriented Systems with Roles. *ACM Transactions on Information Systems*, 14(3), July 1996
14. Mary D. P. Leland, Robert S. Fish, and Robert E. Kraut, Collaborative document production using quilt, *ACM CSCW'88 Proceedings*.
15. Du Li. Home of COCA: Runtime Support and Example Applications, <http://www.cs.ucla.edu/~lidu/coca/>
16. Du Li and Richard R. Muntz, COCA: Collaborative Objects Coordination Architecture, *Proceedings of ACM CSCW '98*, Nov. 1998, Seattle
17. Du Li, Zhenghao Wang, and Richard R. Muntz, "Got COCA?" A New Perspective in Building Electronic Meeting Systems, in *Proceedings of WACC'99 Conference on Work Activities Coordination and Collaboration*, Feb. 1999, San Francisco
18. Du Li, Zhenghao Wang, and Richard R. Muntz, COCA Web: a Generic Platform for World-Wide Collaboration and Electronic Commerce, invited poster, to appear in the *8th Intl. World Wide Web Conference*, May 1999, Toronto
19. Du Li and Richard R. Muntz. Runtime Dynamics in Collaborative Systems. To appear in the *Proceedings of ACM Group'99 International Conference on Supporting Group Work*, Phoenix, Arizona, November 1999.
20. Du Li, Lalita J. Jagadeesan, James D. Herbsleb, and Patrice Godefroid. Verification of Coordination Policies in a Collaborative Framework. To be submitted to the *22nd International Conference on Software Engineering (ICSE'2000)*, Limerick, Ireland, June 2000.
21. Du Li, James D. Herbsleb, Lalita J. Jagadeesan, and Richard R. Muntz. Flexible Awareness Control in Dynamically-Grouped Workspaces. In Preparation.
22. L. Logrippo, M. Faci, and M. Haj-Hussein, An Introduction to LOTOS: Learning by Examples. *Computer Networks and ISDN Systems*, 23, 5, Feb. 1992
23. Steven McCanne, Van Jacobson, *vic*: A Flexible Framework for Packet Video. *ACM Multimedia 1995*
24. Naftaly Minsky and Victoria Ungureanu, Regulated Coordination in Open Distributed Systems, *2nd Intl. Conference on Coordination Languages and Models*, Berlin, Germany, Sept. 1997 Proceedings
25. L. M. Pereira and R. Nasr. Delta-Prolog: a distributed logic programming language. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, 1984
26. J. Rekers and I. Sprinkhuizen, A LOTOS Specification of a CSCW tool. *Proc. of Design of Computer Supported Cooperative Work and Groupware Systems*, Schearding, Austria, 1993
27. Dirk Riehle and Thomas Gross, Role Model Based Framework Design and Integration, in *Proceedings of ACM OOPSLA '98*
28. V. Saraswat and M. Rinard. Concurrent Constraint Programming. In *Proc. of the 17th ACM Symposium on Principles of Programming Languages*, New York, 1999
29. V. Saraswat, M. Rinard, and P. Panangaden. Semantics foundations of concurrent constraint programming. In *Proc. of the 18th ACM Symposium on Principles of Programming Languages*, New York, 1991
30. Ehud Shapiro. Embedding Linda and other joys of concurrent logic programming. Tech. Report, CS-89-07, Dept. of Computer Science, The Weizmann Institute of Science, Rehovot, Israel
31. Ehud Shapiro, The family of concurrent logic programming languages, *ACM Computing Surveys*, 21(3), Sept. 1989
32. Abraham Silberschatz and Peter Galvin. Operating System Concepts. Fifth Edition, p.563-566. Addison Wesley Longman, Inc., 1998
33. Michael VanHilst and David Notkin, Using Role Components to Implement Collaboration-Based Designs, in *Proceedings of ACM OOPSLA '96*
34. R.J. Wieringa, W. De Jong, and P. Sprint, Roles and dynamic subclasses: a modal logic approach. In M. Tokoro and R. Pareschi, editors, *the 8th European Conference on Object-Oriented Programming (ECOOP'94)*, Springer, 1994

Hancock: A Language for Processing Very Large-Scale Data

Dan Bonachea* Kathleen Fisher Anne Rogers Frederick Smith†

*AT&T Labs
Shannon Laboratory
180 Park Avenue
Florham Park, NJ 07932, USA*
bonachea@cs.berkeley.edu
{kfisher,amr}@research.att.com
fms@cs.cornell.edu

Abstract

A signature is an evolving customer profile computed from call records. AT&T uses signatures to detect fraud and to target marketing. Code to compute signatures can be difficult to write and maintain because of the volume of data. We have designed and implemented Hancock, a C-based domain-specific programming language for describing signatures. Hancock provides data abstraction mechanisms to manage the volume of data and control abstractions to facilitate looping over records. This paper describes the design and implementation of Hancock, discusses early experiences with the language, and describes our design process.

1 Introduction

There are many families of programs whose members have a high degree of commonality; such a family is called a *domain*. When the commonality is inherently complex, a domain-specific programming language may help domain programmers develop better software more quickly by factoring the complexity into the language. Recent examples of domain-specific languages and their domains include Envision [SF97] for computer vision, Fran [Ell97] for computer animation, GAL [TMC97] for video card device drivers,

Mawl [ABB⁺97] for dynamic web servers, and Teapot [CRL96, CDR⁺97] for writing coherence protocols. In each case, the domain-specific language moved the burden of writing intricate domain code from the programmer to the compiler (or interpreter). We have designed and built a domain-specific language, called Hancock, to handle the complexity that arises from the scale of *signature* computations [CP98]. As with the earlier languages, Hancock programs are easier to write, debug, read, and maintain than the equivalent programs written in general-purpose programming languages.

Signatures are profiles of customers that are updated daily from information collected on AT&T's network. They are used for a variety of purposes, including fraud detection and marketing. For example, AT&T uses signatures to track the typical calling pattern for each of its customers. If customers far exceed their usual calling levels, the fraud detection system raises alerts. In contrast, a sudden drop in their calling levels signals that they may have chosen another long-distance carrier, triggering a marketing alert. Cortes and Pregibon describe signatures and their uses in more detail [CP98].

At a high level, the computation of a signature is straightforward: process a list of call records and update data in a file based on those records. Unfortunately, performance requirements that arise from the volume of call records and the amount of stored profile data complicate matters substantially. Efficiently coping with the data requires a more

*Now in the EECS department at the University of California at Berkeley.

†Now in the CS department at Cornell University

complex system architecture. Furthermore, the scale pressures programmers to conserve instructions, which tends to reduce program readability. The complexity of the architecture and the code complicates testing, which is already difficult because of long testing cycles.

The scale of signature computations arises from both the number of calls and the number of customers. On a typical weekday, there are more than 250 million calls made on AT&T's network. The call records that are used for signature computations contain only 32 of the more than 200 bytes of data that are collected for each call. To get a sense of the scale of this data, it takes about 15 minutes to read through one weekday's call data (approx 6.5GB) and about two hours to compute a typical signature on one processor of a 16 processor SGI Origin 2000. A typical signature tracks several hundred million phone numbers. Even if we store only two bytes of data per customer, the files are very large, requiring a minimum of 500MB to hold the data. Such files also require significant additional space to provide appropriate indexing.

This scale complicates the architecture of signature programs because at such levels, I/O presents a significant bottleneck. To reduce this bottleneck, signature programs must be structured to minimize I/O. In particular, signature programs sort the call records that they process to improve locality of reference to their signature files, and they cache parts of these files to reduce the number of disk accesses. Both techniques trade improved performance for increased code complexity.

In addition to affecting the high-level architecture, the amount of data complicates the low-level code structure. Programmers writing signature programs have tended to respond to performance pressures by avoiding function calls and thereby writing less modular code. The deeply nested code that results is difficult to decipher, which is a serious problem because it is often the only documentation of the signature it implements. Another complication comes from the fact that the size of the signature files limits the number of bytes we can store per phone number. As a result, we cannot store exact information for each phone number; instead we must approximate. Managing the translation be-

tween the representation that we can afford to store and the representation that we want to compute with complicates the code.

Hancock is a C-based domain-specific language that reduces the coding effort of writing signatures and improves the clarity of the resulting code by factoring into the language all the issues that relate to scale. In particular, Hancock programmers can focus on the signature they want to compute rather than the scaffolding necessary to compute it. The language includes constructs for describing the overall system architecture, for specifying work that should be done in response to events in the call stream, and for specifying the representation of signature data. The Hancock compiler uses these constructs to generate the boiler-plate code that makes hand-written signatures difficult to maintain, without sacrificing efficiency. The Hancock runtime system supports external sorting and efficient access to signature data. Hancock does not address directly the problem of testing; instead, it reduces opportunities for bugs by relieving programmers of the burden of writing intricate code.

This paper describes the design and implementation of Hancock. Section 2 describes the domain in more detail and gives an example signature. Sections 3-5 present Hancock's data and control abstractions. We discuss the implementation of Hancock's compiler and runtime system in Section 6, early experiences with Hancock in Section 7, and our design process in Section 8. We conclude in Section 9.

2 Signature domain

Signatures are a way to associate information with individual telephone numbers. For each phone number, a typical signature contains information about the characteristics of outgoing calls from that number and incoming calls to it. The outgoing data is often split into sub-categories based on the type of call (for example, toll-free, international, intra-state, and other). This section describes the process of computing signatures, discusses the representation of signature data, and presents an example signature.

We first define some basic terminology. Signatures maintain information about phone

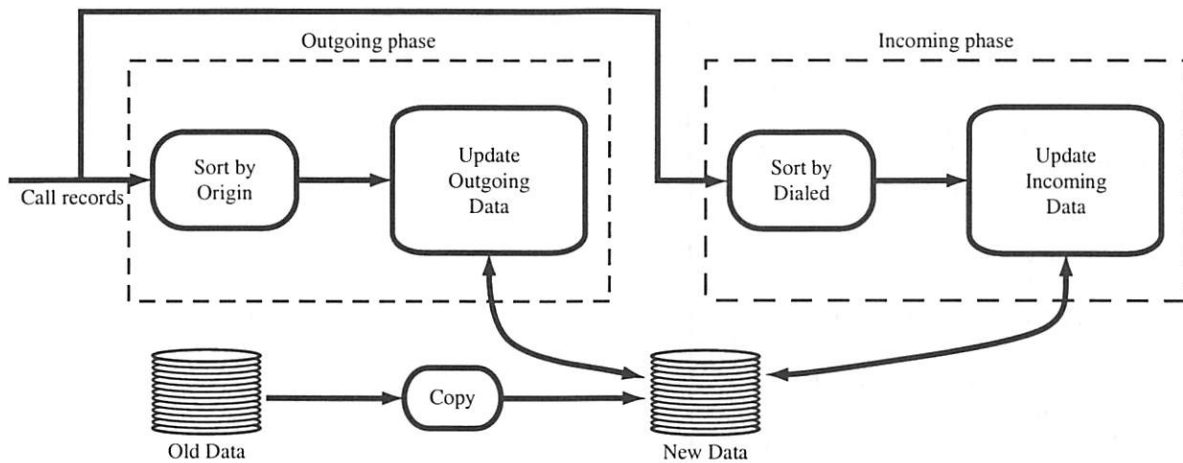


Figure 1: High-level architecture of signature computations

numbers rather than customers. A second database can be used to match signature data with customer information, but that process is outside of Hancock's domain. A telephone number (973 555 1212) contains an *area code* (973), an *exchange* (555), and a *line number* (1212). We often use the term *exchange* to mean the first six digits of a phone number (973 555) and the term *line* to mean the whole phone number. Hancock supports a few basic types for phone numbers, dates, and times: `area_code_t`, `exchange_t`, `line_t`, `date_t`, and `time_t`. It also supports a type for call records:

```

typedef struct {
    line_t origin;
    line_t dialed;
    date_t connectTime;
    time_t duration;
    char isIncomplete;
    char isIntl;
    char isTollFree;
    ...
} callRec_t;

```

Each call record contains two phone numbers: the originating number and the dialed number. Each record also stores the time the call was connected (`connectTime`) and the duration of the call in seconds (`duration`). Call records also contain boolean flags describing the type of call: `isIncomplete`, `isIntl`, `isTollFree`, etc.

2.1 Signature computation

We update signature data daily from the records of the calls made the previous day.

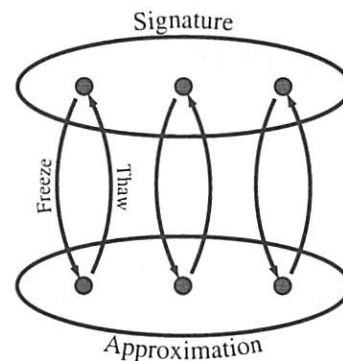


Figure 2: Signature data representation

Figure 1 shows a graphical depiction of the typical architecture we use for such computations. We first sort the call records by the originating phone number and then update the outgoing portion of the signature for each phone number that made a call. We then repeat this process, sorting the call records by the dialed number and updating the incoming portion of the signature for each phone number that received a call. This architecture reduces the I/O needed to process a signature by ensuring good locality for references to the stored signature data at the cost of a pair of external sorts.

2.2 Signature representation

As mentioned earlier, the size of signature files limits the number of bytes we can keep for each phone number. As a result, we cannot keep exact information for each line. In-

stead, we need to approximate. Conceptually, in each signature we have two views for our data: a precise form called the *signature* view and an approximate form called the *approximation* view. We compute with the signature, but store the approximation. The choice of these two views is application specific, as is the method for converting between them. We call the process of converting from the signature to the approximation view *freezing*; the converse process, *thawing*. (See Figure 2.) Note that `thaw(freeze(v))` does not necessarily equal `v` because freezing is often lossy.

Having two views for the data allows programmers to compute with the natural representation while saving disk space, but it comes at the cost of having to manage conversions between the two views.

2.3 Usage signature

This section describes the *Usage* signature, which we use as a running example. Usage approximates cumulative daily call duration for incoming calls, outgoing calls, and outgoing calls to toll-free numbers. Usage uses the same structure to track all three types of calls. In particular, the signature type for each is seconds; the approximation type, a bucket number between zero and fifteen. The buckets represent non-uniform ranges of durations. Bucket zero corresponds to very short calls and serves as the default value for lines with no recorded activity, while bucket fifteen corresponds to long calls. Thawing converts bucket numbers to seconds by associating a default duration with each bucket. Freezing identifies the bucket with the appropriate range of times.

There are two parts to the daily Usage computation. First, we accumulate the precise usage in seconds for a particular phone number for a particular type of call, and then we *blend* that data with the existing signature for that type of call. The code for blending is:

```
blend(new, old) = new*lambda +
                  old*(1-lambda)
```

Blending of this form is common in signature computations.

Pseudo-code for computing part of the Usage signature appears in Figure 3. For brevity, we include only the code for computing the outgoing portion of the signature

```
outgoingUsage(origin, calls)
  int cumTollFree = 0;
  int cumOut = 0;

  uApprox = Get usage data for origin
  uSig = Convert uApprox from buckets
         to seconds

  for c in calls do
    if c.isTollFree then
      cumTollFree += c.duration
    else
      cumOut += c.duration
    endif
  done

  uSig.outTollFree =
    blend(cumTollFree, uSig.outTollFree)
  uSig.out = blend(cumOut, uSig.out)

  uApprox = Convert uSig from seconds
            to buckets.
  Record new uApprox data for origin
```

Figure 3: Pseudo-code for computing part of the Usage signature

for a single line, called *origin* in the pseudo-code. A detailed version of Usage that tracks some additional information can be found in Appendix A.

3 Data model

This section describes Hancock's data model, which includes a model for collections of call records and a model for profile data.

3.1 Call stream

We model a collection of call records as a stream in Hancock. Programmers use the *stream* type operator to declare a new stream type. Such a declaration names the new type and specifies both the *physical* and the *logical* representations of the records in the stream. Intuitively, the physical representation describes the (highly encoded) structure of the records as they exist on disk, while the logical representation describes an expanded form convenient for programming. The declaration specifies a function to convert from encoded physical to expanded logical records. For example, the following code declares a stream type *callStream*:

```

stream callStream {
    getvalidcall : PCallRec_t =>
                    callRec_t;
}

```

For this stream, the physical type is `PCallRec_t`, the logical type is `callRec_t`, and the conversion function `getvalidcall` constructs a logical record from a physical one. Function `getvalidcall` has type

```

char getvalidcall(PCallRec_t *pc
                  callRec_t *c)

```

This function checks that the record `*pc` is valid, and if so, unpacks `*pc` into `*c` and returns `true` to indicate a successful conversion. Otherwise, `getvalidcall` simply returns `false`. Programmers can declare variables of type `callStream` using standard C syntax (for example, `callStream calls`).

We represent streams on disk as a directory that contains binary files. Hancock's wiring-diagram mechanism, which we discuss in Section 5, provides a way to match the name of a directory to a stream.

3.2 Signature data

Hancock provides two mechanisms for describing signature data. Programmers use the `record` declaration to specify the format of a profile and the `map` declaration to specify the mapping between phone numbers and profiles.

Records are designed to capture the relationship depicted in Figure 2. They specify the types for the signature and approximation views of a profile, as well as the freeze and thaw expressions for converting between these types. We use the following simple record to introduce the pieces of a `record` declaration.

```

record uField(uSig, uApprox){
    int <=> char;
    uSig(b) = bucketToSec[b];
    uApprox(s) = secToBucket(s);
}

```

This declaration introduces three types:

- `uField`: the type of the record,
- `uSig`: the type of the left-hand view (`int`), and
- `uApprox`: the type of the right-hand view (`char`).

The `uSig(b) = ...` portion of the record declaration specifies how to thaw `uApprox b` to produce a `uSig` value. Similarly, `uApprox(s) = ...` specifies how to freeze a `uSig s` to obtain a `uApprox` value. In this application, `uApprox(s)` uses the function `secToBucket` to convert the seconds stored in integer `s` into a bucket number, and `uSig(b)` uses the array `bucketToSec` to convert a bucket stored in `b` into the corresponding mean number of seconds for that bucket. Together, these expressions are an example of where `thaw(freeze(s))` does not equal `s`.

Records can have more than one field (in which case the fields are named), and they can be included in other record declarations. For example, a second `record` declaration that appears in the Usage signature, `uLine`, has the following form:

```

record uLine(uSig, uApprox){
    uField in;
    uField out;
    uField outTF;
}

```

As in our earlier example, this declaration introduces three types: `uLine`, `uSig`, and `uApprox`. The type `uLine` is the type of the record. The types `uSig` and `uApprox` are equivalent to C structures constructed from the left and right types of `uField`:

```

typedef struct {
    int in;
    int out;
    int outTF;
} uSig;

typedef struct {
    char in;
    char out;
    char outTF;
} uApprox;

```

Because the record `uLine` does not include explicit freeze and thaw expressions, Hancock constructs them automatically from the freeze and thaw expressions of the record's fields. For this record, the compiler constructs the following freeze function:

```

uApprox freeze(uSig s){
    uApprox a;
    a.in = secToBucket(s.in);
    a.out = secToBucket(s.out);
    a.outTF = secToBucket(s.outTF);
    return a;
}

```

The thaw function is constructed similarly.

Although this example record only contains fields with record types, fields may also have regular C types. In this context, C types can be thought of as records with the same left- and right-hand type and the identity function for freezing and thawing.

To convert between views, Hancock provides the view operator (\$). The expression `ua$uSig` converts `uApprox ua` to a `uSig`, using the conversion specified implicitly in the `uLine` declaration. Expression `us$uApprox` behaves analogously.

Hancock's `map` declaration provides a way to associate data with keys. Typically a map does not contain data for every possible key. Consequently, Hancock supports the notion of a *default value*, which is returned when a programmer requests data for a key that does not have a value stored in the map. For example, in the following `map` declaration, the keys have type `line_t`, the data are structures of type `uApprox`, and the default value is the constant `uApprox` structure consisting of all zeros.

```
map uMap {
  key line_t;
  value uApprox;
  default {0,0,0};
}
```

Defaults may also be specified as functions that use the key in question to compute an appropriate default for that key. For example, the `map` declaration below specifies a function, `lineToDefault`, to call with the line in question when a default record is needed.

```
map uMapF {
  key line_t;
  value uApprox;
  default lineToDefault;
}
```

A common use for this mechanism is to construct defaults by querying another data source.

The identifier `uMap` names a new map type. Variables of this type can be declared using the usual C syntax (for example, `uMap usage`). Hancock provides an indexing operator `<:pn:>` to access values in a map. The code:

```
line_t pn;

u = usage<:pn:>;
...
usage<:pn:> = u;
```

gives an example of reading from and writing to a map. The usual idiom for accessing map data combines the indexing and view operators as in:

```
us = usage<:pn:>$uSig;
```

Hancock also provides an operator `:=:` to copy maps. In particular, the statement

```
new_usage :=: usage;
```

causes `uMap` map `new_usage` to be initialized with the data from `usage`.

3.3 Discussion

By providing the programmer with appropriate abstractions, Hancock reduces the intellectual burden of writing signatures. Although programmers were freezing and thawing their data prior to Hancock, they had not abstracted this idea. The result was numerous bugs caused by confusing the types of the two views. The structure enforced by records eliminates many of these bugs by requiring programmers to document the relationship between the two views, and to apply the view operator to convert between them explicitly. As an added benefit, records simplify signature code by generating conversion functions automatically from record fields when possible.

Maps provide an efficient implementation for the most performance critical part of signature programs. The index operation is more convenient than a library interface, and it provides stronger type-checking.

4 Computation Model

Hancock's computation model is built around the notion of iterating over a sorted stream of calls. Sorting call records ensures good locality for references to the signature data that follow the sorting order. Off-direction references may not have good locality, however. For example, if we sort the call records by the originating number, then updating the usage for that number would have

good locality, while updating the usage for the dialed number would suffer from bad locality. Consequently, signature computations are typically done with multiple passes over the data, each sorting the data in a different order and updating a different part of the profile data. We call each such pass, represented in Figure 1 as a dashed box, a *phase*.

A phase starts by specifying a name and a parameter list. The body of a phase has three pieces: an iterate clause, a list of variable declarations, and a list of event clauses. The following pseudo-code outlines the outgoing phase of the usage signature:

```
phase out(callStream calls, uMap usage) {
    iterate clause
    variable declarations
    event clauses
}
```

This code defines a phase `out` that takes two parameters: a stream of calls and a usage map. The iterate clause specifies an initial stream and a set of transformations to produce a new stream. Variables declared at the level of a phase are visible throughout that phase. The event clauses specify how to process the transformed stream. The next two subsections describe these clauses.

4.1 Iterate Clause

Through the iterate clause, Hancock allows programmers to transform a stream of logical call records into a stream of records tailored to the particular signature computation. The iterate clause has the following form:

```
iterate
    over stream variable
    sortedby sorting order
    filteredby filter predicate
    withevents event list
```

We explain each of these pieces in turn. The `over` clause names an initial stream to transform. The `sortedby` clause specifies a sorting order for this stream. For example, the calls could be sorted by the originating phone number or by the connect time. At present, the allowable sorting orders are hard-coded into Hancock. The `filteredby` clause specifies a predicate that is used to remove unneeded records from the stream. For example, a call stream may include incomplete calls, which are not used by the Usage signature. Removing unneeded call records before

processing the stream simplifies the processing code.

The `withevents` clause specifies which events are relevant to this signature. We call the occurrence of a group of calls in the stream an *event*. Depending on the sorting order, different groups of calls can be identified in the call stream. For example, if the calls are sorted by originating number, the possible groups are:

- calls for the same area code,
- calls for the same exchange,
- calls for the same line, or
- a single call.

Within an event there are two important sub-events: the beginning of the block of calls and its ending. The possible events and sub-events are related hierarchically; calls are nested with lines, lines within exchanges, and exchanges within area codes.

Putting all these pieces together, the iterate clause for the outgoing phase of Usage has the following form:

```
iterate
    over calls
    sortedby origin
    filteredby noIncomplete
    withevents line, call;
```

This code specifies that `calls` is the initial stream, that it should be sorted by the originating number, that it should be filtered using function `noIncomplete`, which removes incomplete calls from the stream, and that two events, `line` and `call`, are of interest.

4.2 Event Clauses

The iterate clause specifies the events of interest in the stream, but it does not indicate what to do when an event is detected. The *event clauses* of a phase specify code to execute in response to a given event. We illustrate this structure in Figure 4. Programmers supply event code for the boxes, while Hancock generates the control-flow code that corresponds to the arrows.

Each event has a name that corresponds to the event name listed in the iterate clause. Each event takes as a parameter the portion of the call record that triggered the event.

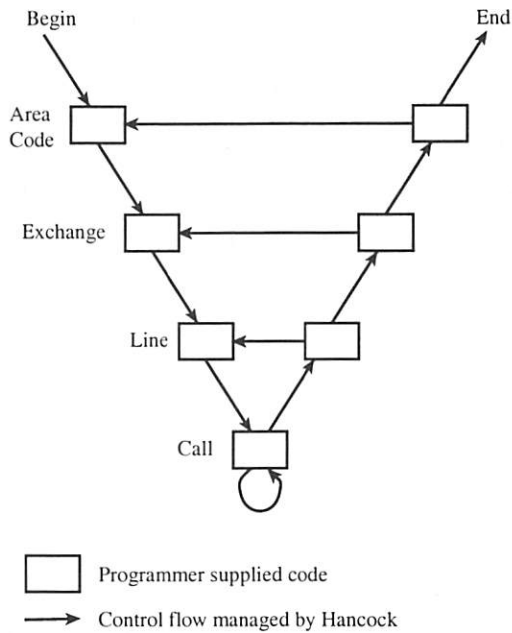


Figure 4: Hierarchical event structure

For example, an area code event is passed the area code shared by the block of calls that triggered the event. The body of each event has three parts: a list of variable declarations, a begin sub-event, and an end sub-event. Variables declared at the level of an event are shared by both its sub-events and are available to events lower down in the hierarchy, using a scoping operator (**event name::variable name**). A common pattern is for the programmer to declare a variable in the code for a line event, to initialize it in the begin-line sub-event, to accumulate information using that variable while processing calls, and then to store the accumulated information during the end-line sub-event code.

A sub-event is a kind (**begin** or **end**) followed by a C-style block that contains a list of variable declaration and Hancock and C statements. Variables declared in a sub-event are visible only within that sub-event. The code in Figure 5 implements the line and call events for the outgoing phase of Usage.

Note that the **call** event does not have sub-events. We call such events *bottom-level* events. The lowest event in any list of events is a bottom-level event. For example, Frequency, a signature that we discuss later, does not collect summary information for a set of calls. It tracks only the existence of at least one call, so **line** is its bottom-level event.

```

event line(line_t pn) {
    uSig cumSec;

    begin {
        cumSec.out = 0;
        cumSec.outTF = 0;
    }

    /* process calls */

    end {
        uSig us;

        us = usage<:pn:>$uSig;
        us.outTF =
            blend(cumSec.outTF, us.outTF);
        us.out = blend(cumSec.out, us.out);
        usage<:pn:> = us$uApprox;
    }
}

event call(callRec_t c) {
    uSig line::cumSec;

    if (c.isTollFreeCall)
        cumSec.outTF += c.duration;
    else
        cumSec.out += c.duration;
}
  
```

Figure 5: Event code for Usage signature

4.3 Discussion

Hancock's event specifications have several advantages. First, the specifications have the flavor of function definitions with their attendant modularity advantages, but without their usual cost because the Hancock compiler expands the event definitions in-line with the control-flow code. Second, having the compiler generate the control flow removes a significant source of bugs and complexity from Hancock programs. Finally, programmers can use the variable-sharing mechanism to share information across events.

A common question when thinking about designing a domain-specific language is whether or not a library would suffice. We rejected the library option largely because of Hancock's control-flow abstractions. In particular, expressing Hancock's event model and the information sharing it provides proved awkward in a call-back framework, the usual technique for implementing such abstractions.

5 Wiring Diagram

In the previous section, we explained that computing a signature may require multiple passes over the data. Hancock provides the `sig_main` construct to express the data flow between passes and to connect command-line input to the variables in the program. The arcs between the phase boxes in Figure 1 depict this construct. The following code implements `sig_main` for Usage:

```
void sig_main(
    const callStream calls <c:>,
    exists const uMap y_usage <u:>,
    new      uMap usage <U:>) {
    usage := y_usage;
    out(calls, usage);
    in(calls, usage);
}
```

There are three parameters to the Usage signature. The first is a stream that contains the raw call data. The `const` keyword indicates that this data is read-only. The syntax (`<c:>`) after the variable name `calls` specifies that this parameter will be supplied as a command-line option using the `-c` flag. The colon indicates that this flag takes an argument, in this case the name of the directory that holds the binary call files. The absence of a colon indicates that the parameter is a boolean flag. The Hancock compiler generates code to parse command-line options. The second parameter is a Usage map, the name of which is specified using the `-u` flag. The `const` qualifier indicates the map is read-only, while the `exists` annotation indicates the map must exist on disk. The final parameter names the Usage map used to hold the result of this signature computation; the `-U` flag specifies the file name for this map. The `new` qualifier indicates that the map must not exist on disk.

In general, the body of `sig_main` is a sequence of Hancock and C statements. In Usage, `sig_main` copies the data from `y_usage` into `usage` and then invokes Usage's outgoing and incoming phases with the raw call stream and the Usage map under construction as arguments.

5.1 Discussion

The wiring diagram clarifies the dataflow between phases. For example, some signatures need to make off-direction references to

signature data. A question that arises is: are these references referring to data computed in a previous phase or to data computed the previous day? This question can be answered by looking at `sig_main`. If the parameters to the phase do not include the original input map, then all references must be to the partially computed map.

The automatic generation of argument parsing code is convenient and removes a source of tedium, but its real benefit is that it connects Hancock variables to their on-disk counterparts. It helps programmers protect valuable data through the `const`, `new` and `exists` qualifiers. The runtime system catches attempts to write to constant data and generates error messages.¹ It detects when data annotated as `new` already exists or when data tagged with `exists` is not on disk, in each case reporting a run-time error. These data-protection features are important when it is time-consuming or even impossible to reconstruct an accidentally overwritten signature.

6 Implementation

Our implementation of Hancock consists of a compiler that translates Hancock code into plain C code, which is then compiled and linked with a runtime system to produce executable code. We modified CKIT, a C-to-C translator written in ML[SCHO99] to parse Hancock and translate the resulting extended parse tree into abstract syntax for plain C. The compiler generates code for the various Hancock operators and clauses and for the `main` routine. The runtime system, which is written in C, manages the representation of Hancock data on-disk and in memory. It converts between these representations as necessary and it mediates all access to the data.

We were able to build Hancock relatively quickly by leveraging other people's software. In particular, we built the compiler on top of CKIT, an existing C-to-C translator[SCHO99] written for the purpose of building compilers for C-based domain-specific languages. We also used a collection of libraries: `msocket`[Lin99, MMB92], `sfio`[KV91], and `vmalloc`[Vo96]. `Msort` pro-

¹We intend to check for writes to `const` data at compile time eventually.

Table 1: Example signatures.

Signature	Description
Usage	Average daily usage
Frequency	Calling frequency
Activity	Days since last seen
Bizocity	"Business-likeness"

vides an external sorter and sfio supports 64-bit files, making these two libraries particularly useful in addressing issues of scale.

7 Early experiences

Table 1 briefly describes four signatures: Usage, Frequency, Activity, and Bizocity. These signatures are computed daily from call records. In this section, we discuss how these signatures use Hancock's data and control-flow mechanisms.

The maps used by these signatures all have index types of `line_t` and value types that are records. Usage, Frequency, and Activity use constant defaults. Bizocity uses a default function that queries a secondary map, which indicates whether the phone is a known residence, a known business, or unknown.

The records used in these signatures vary based on the application, but they all have a common form: the desired profile contains several fields that have the same underlying structure. To express this structure in Hancock, we use two records: one to describe the basic fields and a second to group these fields into a profile.

Table 2 describes the basic fields for the sample signatures and indicates how many such fields are contained in the profile record. In all these examples, the approximation type is a range that can be represented with a C `char` type. The signatures use different approximation techniques. *Bucketing* divides the range of signature values into disjoint buckets and associates a default value with each such bucket. With this technique, freezing converts a signature value into the containing bucket, whereas thawing returns the default value for a bucket. Bucketing can use either fixed-width or variable-width buckets. *Clamping* converts values above the range of

signature values to the largest value in the range and values below the range to the lowest value in the range.

In all four signatures, the amount of Hancock code needed to describe the data is small. The largest, Bizocity, takes fewer than 30 lines.

In terms of control-flow, the example signatures share the same high-level structure, each containing two phases: one to compute information for outgoing calls and another for incoming calls. The event structures for the signatures are different, however. Frequency tracks only the existence of a call for a given number, so its bottom-level event is the line event. Activity and Usage do work at both call and line events. Bizocity uses these events and does significant computation at the exchange level.

The Hancock code that implements these phases is small: the smallest, Frequency, takes 40 lines of code; the largest, Bizocity, takes 300 lines, more than 100 of which are for processing exchange events.

In all, these examples indicate that Hancock data descriptions are compact and that the event processing code is modest in size. Ideally, we would like to compare the Hancock implementations with hand-written C implementations. Unfortunately, this comparison is very hard to do fairly. The only C implementation of Usage, Activity, and Bizocity available to us is a program that combines the computation of all three signatures and has code to manage the on-disk representations of the signature files embedded in it. This program is about 1500 lines of code.

8 Design Process

This section describes the process that we used in the design of Hancock and discusses the lessons we learned from our experience. Designing a domain specific language involves developing a model of the domain and then embodying that model in a language. The language should capture the commonalities of the domain effectively and let the domain experts worry about the details specific to an individual application. Figure 6 depicts the process we followed; it can be viewed as an elaboration of the first box of the FAST process [GJKW97].

Table 2: Record structure for example signatures

Signature	Signature type	Approximation type	Approximation method	Number of fields
Usage	int	(0-15)	variable-width buckets	4
Frequency	double	(0-255)	fixed-width buckets	2
Activity	int	(0-39)	clamping	3
Bizocity	char	(0-15)	fixed-width buckets	2

First, we alternated talking with domain experts about sample signatures and constructing models of the domain that captured the common elements of these signatures. By iterating, we got feedback on the models and developed a common vocabulary.

Once we had a good model for signatures, we applied this model by hand to construct a few sample applications. This experience led us to replace our model with one based on more primitive abstractions because the original abstractions were too high-level to serve as the basis for a programming language. We eventually used these hand-written signatures to establish that the code we expected to generate automatically would perform adequately and to evaluate whether a library-based solution was sufficient.

After we had revised our model based on the hand-written signatures, we translated it into an actual language design. This translation was not straightforward because although the model revealed what concepts needed to be expressible in the language, it did not give much guidance as to how they should be expressed. Then we wrote sample applications using our design, evaluated the resulting programs, and revised the design as appropriate. We found that we needed to iterate through this process many times. We discussed these sample applications with the domain experts to confirm that we had captured the essential elements of the domain.

Finally, we implemented a compiler and runtime system for Hancock and evaluated the signatures written in Hancock. This evaluation led us to refine many aspects of the original design, including the stream model, the view operator, the `sig_main` annotations `new` and `exists`, support for user-supplied compression functions, the `record` construct, and the implementation of maps.

8.1 Lessons Learned

Our goal in this project was to design and implement a language that would be adopted by signature researchers. While this project is not yet finished and so we cannot claim complete success at this time, we believe that we are on the right path because the signature researchers have become advocates for Hancock. We believe that three things that we did were responsible for our success and could be useful to others who want to design domain specific languages.

First, we were open to constantly revising our models based on many different kinds of input. Figure 6 highlights the many sources of feedback that we employed in our design process. In addition to consulting with the domain experts at many points in the process, we also built artifacts at several intermediate stages. These artifacts proved useful even though they are not the final product of our work.

Second, we worked closely with domain experts and valued their input. This point deserves elaboration, as it seems quite obvious that language designers should heed their experts. The fact that domain experts may not be expert programmers leads to a temptation to dismiss their comments. Sample implementations they provide may be written badly. Consequently, it is easy to leap to the erroneous conclusion that they do not know what they are doing. For example, signature researchers had developed a map representation that drew a lot of criticism from others outside the domain. When we investigated their representation, we learned that although their implementation had some problems, the basic structure provided very good random access time. Such access is crucial to their clients, but it had been ignored by other onlookers. As language designers, we

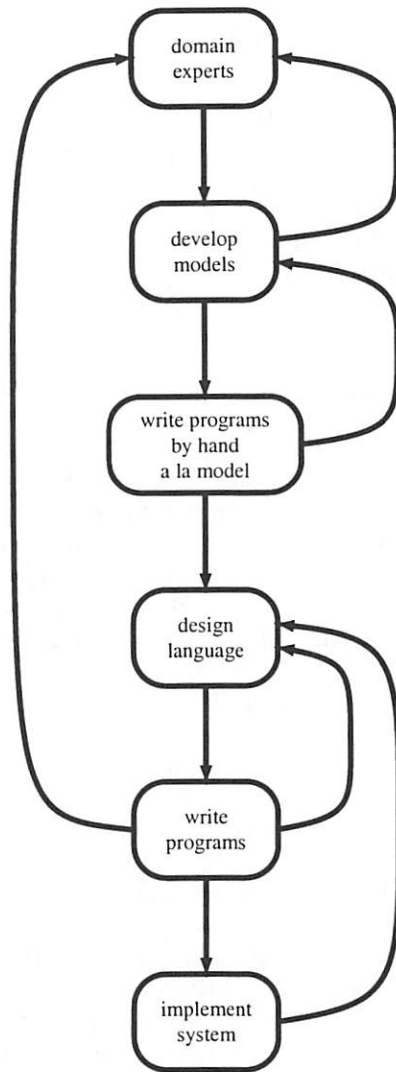


Figure 6: Design Process

had to be very careful to separate crucial domain constraints from irrelevant details in the existing implementations.

Finally, gaining credibility with domain experts was essential because they are the potential users for the language. By designing concrete models in response to our discussions with them, we established that we were serious about helping them and that we understood their domain. It also gave us a focus for our discussions. By handwriting signatures in C, we established that we could satisfy their performance requirements. By choosing C as the basis for Hancock, we kept Hancock close to their usual programming environment. By using signature representations consistent with the ones the domain

experts had designed, we demonstrated that Hancock programs could manipulate their existing data without difficulty. Most importantly, by consulting with the domain experts at every point, we improved the design and got them excited about using Hancock.

9 Conclusions

Hancock handles the scale of the data used in signature computations, thereby reducing coding effort and improving the clarity of signature code. The language, compiler, and runtime system provide the scaffolding necessary to compute signatures, leaving programmers free to focus on the signatures themselves. In particular, Hancock's wiring diagram makes the relationships among phases clear. Its event model allows programmers to specify the work needed to compute a signature without having to write complicated control-flow code by hand. Finally, Hancock's data model makes writing signatures less error-prone by handling multiple data representations automatically.

We plan to extend Hancock in two ways. First, we intend to enrich the set of operations that Hancock provides for streams. For example, we plan to add a reduction operation that would allow programmers to preprocess streams to combine related records. Second, we intend to broaden the class of data that can be processed using Hancock, for example, to include Internet protocol logs or billing records. To accomplish this goal, we need to provide a mechanism for describing data streams. Such a description must include how such streams can be sorted and how to detect events based on the sorting order.

10 Acknowledgments

We would like to thank Corinna Cortes and Daryl Pregibon for their help in understanding the signature domain, Glenn Fowler, John Linderman, and Phong Vo for their assistance with the run-time system, Nevin Heintze and Dino Oliva for their help in using CKIT, Dan Suciu and Mary Fernandez for discussions which helped refine the stream model, and John Reppy for producing the pictures.

References

- [ABB⁺97] Atkins, D., T. Ball, M. Benedikt, G. Bruns, K. Cox, P. Mataga, and K. Rehor. Experience with a domain specific language for form-based services. In *Proceedings of the USENIX '97 Conference on Domain-Specific Languages*, 1997.
- [CDR⁺97] Chandra, S., M. Dahlin, B. Richards, R. Y. Wang, T. E. Anderson, and J. R. Larus. Experience with a language for writing coherence protocols. In *Proceedings of the USENIX '97 Conference on Domain-Specific Languages*, 1997.
- [CP98] Cortes, C. and D. Pregibon. Giga mining. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, 1998.
- [CRL96] Chandra, S., B. Richards, and J. R. Larus. Teapot: Language support for writing memory coherence protocols. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, 1996.
- [Ell97] Elliott, C. Modeling interactive 3D and multimedia animation with an embedded language. In *Proceedings of the USENIX '97 Conference on Domain-Specific Languages*, 1997.
- [GJKW97] Gupta, N. K., L. J. Jagadeesan, E. E. Koutsofios, and D. M. Weiss. Auditdraw: Generating audits the fast way. In *Proceedings of the Third IEEE Symposium on Requirements Engineering*, 1997.
- [KV91] Korn, D. G. and K.-P. Vo. SFIO: Safe/fast string/file IO. In *Proc. of the Summer '91 Usenix Conference*. USENIX, 1991, pp. 235–256.
- [Lin99] Linderman, J. Msort. Private communication, 1999.
- [MMB92] McIlroy, M. D., P. M. McIlroy, and K. Bostic. Engineering radix sort. *Technical Memorandum 11260-920902-23TMS*, AT&T Bell Labs, Murray Hill, NJ, September 1992.
- [SCHO99] Siff, M., S. Chandra, N. Heintze, and D. Oliva. Pre-release of C-frontend library for SML/NJ. See <http://cm.bell-labs.com/cm/cs/what/smlnj/index.html>, 1999.
- [SF97] Stevenson, D. E. and M. M. Fleck. Programming language support for digitized images or, The monsters in the closet. In *Proceedings of the USENIX '97 Conference on Domain-Specific Languages*, 1997.
- [TMC97] Thibault, S., R. Marlet, and C. Consel. A domain specific language for video device drivers: From design to implementation. In *Proceedings of the USENIX*

'97 Conference on Domain-Specific Languages, 1997.

- [Vo96] Vo, K.-P. Vmalloc: A general and efficient memory allocator. *Software—Practice and Experience*, **26**, 1996, pp. 1–18.

A The Usage signature

```
#define NUMBINS 16
int bucketToSec[NUMBINS]= { ... };
char secToBucket(int v) { ... };

record uField(ufSig, ufApprox) {
    int <=> char;
    ufSig(b) = bucketToSec[b];
    ufApprox(s) = secToBucket(s);
}

record uLine(uSig, uApprox) {
    uField in;
    uField out;
    uField outTF;
    uField outInt1;
}

map uMap {
    key line_t;
    value uApprox;
    default {0,0,0};
}

#define LAMBDA .15
#define blend(new, old) \
    (((new) * LAMBDA) + \
     ((old)*(1 - LAMBDA)))

#include calls.h
int getvalidcall(PCallRec_t *pc,
                 callRec_t *c){...}
stream callStream
    {getvalidcall: pCallRec_t =>
     callRec_t}

char noInt10rIncomplete(callRec_t *c) {
    return !(c->isIncomplete) &&
           !(c->isInt1);
}

char noIncomplete(callRec_t *c) {
    return !(c->isIncomplete);
}
```

```

phase out(callStream calls,
          uMap usage) {

```

```

    iterate
    over calls
    sortedby origin
    filteredby noIncomplete
    withevents line, call;

```

```

event line(line_t pn) {
    uSig cumSec;

```

```

    begin {
        cumSec.outTF = 0;
        cumSec.outIntl = 0;
        cumSec.out = 0;
    }

```

```

    end {
        uSig us = usage<:pn:>$uSig;
        us.outTF =
            blend(cumSec.outTF, us.outTF);
        us.outIntl =
            blend(cumSec.outIntl, us.outIntl);
        us.out =
            blend(cumSec.out, us.out);
        usage<:pn:> = us$uApprox;
    }

```

```

}

```

```

event call(callRec_t c) {
    uSig line::cumSec;

```

```

    if (c.isTollFree)
        cumSec.outTF += c.duration;
    else if (c.isIntl)
        cumSec.outIntl += c.duration;
    else
        cumSec.out += c.duration;
} /* end call event */

```

```

}/* end out phase */

```

```

phase in(callStream calls,
          uMap usage){

```

```

    iterate
    over calls
    sortedby dialed
    filteredby noIntlOrIncomplete
    withevents line, call;

```

```

event line(line_t pn) {
    uSig cumSec;

```

```

    begin {
        cumSec.in = 0;
    }

```

```

    end {
        uSig us = usage<:pn:>$uSig;
        us.in =
            blend(cumSec.in, us.in);
        usage<:pn:> = us$uApprox;
    }

```

```

}

```

```

event call(callRec_t c) {
    uSig line::cumSec;

```

```

    cumSec.in += c.duration;
} /* end call event */

```

```

}/* end in phase */

```

```

void sig_main(

```

```

    const callStream calls <c:>) {
    exists const uMap y_usage <u:>,
    new          uMap usage <U:>

```

```

        usage :=: y_usage;
        out(calls, usage);
        in(calls, usage);
    }

```

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits:

- Free subscription to *login*, the Association's magazine, published eight-ten times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and C++, book and software reviews, summaries of sessions at USENIX conferences, and Snitch Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, via the USENIX Online Library on the World Wide Web.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as object-oriented technologies, security, operating systems, electronic commerce, and NT – as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Discount on BSDI, Inc. products.
- Discount on all publications and software from Prime Time Freeware.
- Savings (10-20%) on selected titles from Academic Press, Morgan Kaufmann, New Riders/Cisco Press/MTP, O'Reilly & Associates, OnWord Press, The Open Group, Sage Science Press, and Wiley Computer Publishing.
- Special subscription rates for Cutter Consortium newsletters, *The Linux Journal*, *The Perl Journal*, *IEEE Concurrency*, *Server/Workstation Expert*, *Sys Admin Magazine*, and all Sage Science Press journals.

Supporting Members of the USENIX Association:

C/C++ Users Journal	JSB Software Technologies	O'Reilly & Associates Inc.
Cirrus Technologies	Lucent Technologies	Performance Computing
Cisco Systems, Inc.	Macmillan Computer Publishing,	Questa Consulting
CyberSource Corporation	USA	Sendmail, Inc.
Deer Run Associates	Microsoft Research	Server/Workstation Expert
Greenberg News Networks/MedCast	MKS, Inc.	TeamQuest Corporation
Networks	Motorola Australia Software Centre	UUNET Technologies, Inc.
Hewlett-Packard India	NeoSoft, Inc.	Web Publishing, Inc.
Software Operations	New Riders Press	Windows NT Systems Magazine
Internet Security Systems, Inc.	Nimrod AS	WITSEC, Inc.

Sage Supporting Members:

Atlantic Systems Group	Macmillan Computer Publishing,	O'Reilly & Associates Inc.
Collective Technologies	USA	Remedy Corporation
D. E. Shaw & Co.	Mentor Graphics Corp.	RIPE NCC
Deer Run Associates	Microsoft Research	SysAdmin Magazine
Electric Lightwave, Inc.	MindSource Software Engineers	Taos Mountain
ESM Services, Inc.	Motorola Australia Software Centre	TransQuest Technologies, Inc.
GNAC, Inc.	New Riders Press	Unix Guru Universe

For further information about membership, conferences or publications, contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA.

Phone: 510-528-8649. Fax: 510-548-5738.

Email: office@usenix.org.

URL: <http://www.usenix.org>.

